iOS ドキュメントベース アプリケーション プログラミングガイド

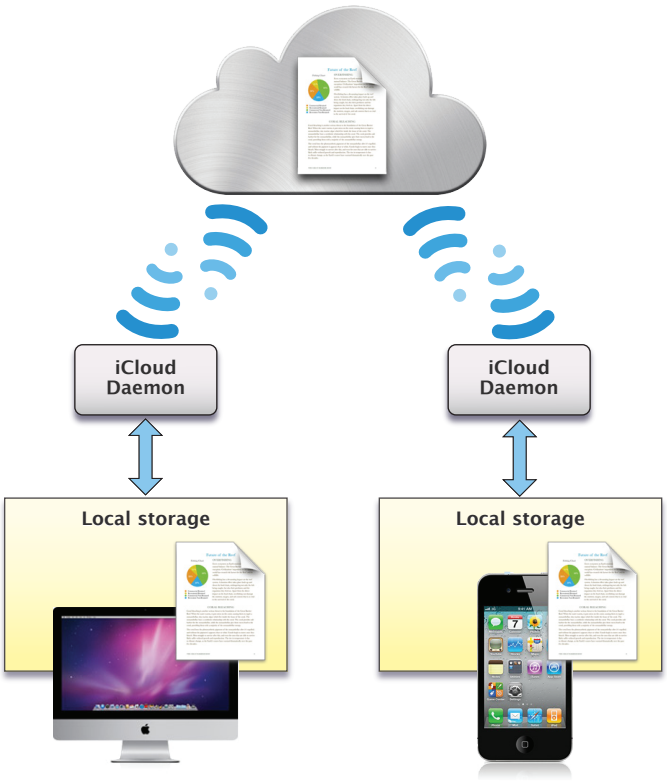
Developer

目次

図、表、リスト

iOS用のドキュメントベースアプリケーションについて

UIKitは、多数のドキュメントを管理するアプリケーションの開発を支援するフレームワークです。各ドキュメントは独自形式のデータ群から成り、アプリケーションサンドボックス内、あるいはiCloud上のファイルに格納されます。



この支援機能の中核となるのが、iOS 5.0で導入された`UIDocument`クラスです。ドキュメントベースアプリケーションは、`UIDocument`のサブクラスを作成して、ドキュメントデータをメモリ上のデータ構造の形でロードし、また、ドキュメントファイルに書き込むデータを`UIDocument`に渡すことになります。`UIDocument`にはドキュメント管理に関するさまざまな処理機能が組み込まれており、開発者がコードを記述する必要はありません。`UIDocument`は、iCloudと統合されていることはもとより、ドキュメントデータをバックグラウンドで読み書きするようになっているため、その間アプリケーションが応答しなくなる、ということもありません。また、ドキュメントデータを定期的に自動保存するので、ユーザは意識的に保存操作をする必要がなくなります。

At a Glance

ドキュメントベースアプリケーションはさまざまなことに気を配らなければなりませんが、その開発は、決して難しい作業ではありません。

ドキュメントオブジェクトはモデルコントローラである

ドキュメントオブジェクト（`UIDocument`のサブクラスのインスタンス）は、「モデル-ビュー-コントローラ」デザインパターンでいうモデルコントローラとして働きます。ドキュメントオブジェクトが管理するのは、ドキュメントに結びつけられたデータ、具体的には、ユーザが閲覧、編集するものを内部的に表現するモデルオブジェクトです。その一方で、ドキュメントオブジェクトは通常、ユーザに向けてドキュメントを表示する、ビューコントローラに管理されます。

関連する章:「ドキュメントベースアプリケーションの設計」（10 ページ）

アプリケーション設計時にドキュメントデータ形式その他の問題を検討する

実際にコードを記述する前に、ドキュメントベースアプリケーションに特有の、さまざまな設計上の問題点を検討しておかなければなりません。最も重要なのは、アプリケーションにとって、どのようなドキュメントデータ形式が最適であるか、どのような形式にすればiOSとMac OS Xの*両方*でうまく扱えるか、ということです。どのようなドキュメントタイプが適しているでしょうか。

さらに、ビューコントローラ（およびビュー）についても検討が必要です。ドキュメントを開く、エラー箇所を表示する、ドキュメントをiCloudストレージに、またはiCloudストレージから移動する、などの処理を管理するところです。

関連する章:「ドキュメントベースアプリケーションの設計」（10 ページ）、「ドキュメントベースアプリケーションの概要」（19 ページ）

UIDocumentのサブクラスを作成し2つのメソッドをオーバーライドする

ドキュメントオブジェクトが主として担うのは、ドキュメントファイルと、内部的にドキュメントデータを表現するモデルオブジェクトの間を結び、データをやり取りする「導管」の役割です。そのため`UIDocument`クラスには、データをドキュメントファイルに書き出す機能、ファイルから読み込んだデータをもとにモデルを初期化する機能が必要です。この役割を組み込むため、`UIDocument`のサブクラスに、`contentsForType:error:`メソッド、`loadFromContents:ofType:error:`メソッドをそれぞれオーバーライドする必要があります。

関連する章:「独自のドキュメントオブジェクトの作成」（25 ページ）

アプリケーションはライフサイクル全体を通してドキュメントを管理する

アプリケーションは、ドキュメントのライフサイクル全体に渡り、次のようなイベントを管理します。

- ドキュメントを作成する
- ドキュメントを開く/閉じる
- ドキュメントの状態変化を監視し、エラーや版の食い違いに対処する
- ドキュメントをiCloudストレージに移動する（iCloudストレージから削除する）
- ドキュメントを削除する

関連する章: 「ドキュメントのライフサイクル管理」 （32 ページ）

アプリケーションはユーザの要求に応じ、ドキュメントファイルをiCloudに格納する

ユーザには、ドキュメントファイルをすべてiCloudストレージに置くか、ローカルサンドボックスに保存するか、の選択肢が与えられます。ドキュメントファイルをiCloudに移動するためには、ドキュメントの格納場所（アプリケーションのiCloudコンテナディレクトリ内）を表すファイルURLを組み立て、このファイルURLを渡して`NSFileManager`クラスの適当なメソッドを呼び出します。iCloudストレージからアプリケーションサンドボックスに移動する処理も、その手順は同様です。

関連する章: 「ドキュメントのライフサイクル管理」 （32 ページ）

アプリケーションはドキュメントデータを自動的に保存する

`UIDocument`は「保存操作なし」モデルに基づき、所定の間隔でドキュメントデータを自動保存します。通常、ユーザが意識的にドキュメントを保存する必要はありません。しかし、この「保存操作なし」モデルでも問題が生じないよう、取り消し/やり直しの処理を実装する、ドキュメントの状態変化を追跡するなど、アプリケーション側も役割の一部を担う必要があります。

関連する章: 「状態変化の追跡、取り消し処理」 （51 ページ）

アプリケーションは版の食い違いを解決する

ドキュメントをiCloudに格納すると、ドキュメントに版の食い違いが発生することがあります。この場合、UIKitはアプリケーションにその旨を通知します。アプリケーションは、この食い違いを自動的に解決するか、あるいは採用する版をユーザに問い合わせる必要があります。

関連する章: 「ドキュメントの版の食い違い解消」 （54 ページ）

How to Use This Document

ドキュメントベースアプリケーションのコードを実際に書き始める前に、少なくとも最初の2章、「ドキュメントベースアプリケーションの設計」（10 ページ）と「ドキュメントベースアプリケーションの概要」（19 ページ）だけは読んでおいてください。この章では、設計や構成に関する問題点を示し、高品質のドキュメントベースアプリケーション開発に必要な作業を概観しています。

Prerequisites

この資料『*Document-Based Application Programming Guide for iOS*』を読む前に、『*iOS App Programming Guide*』、特にiCloud統合に関する内容に目を通しておいてください。

関連項目

『*Document-Based Application Programming Guide for iOS*』の関連資料を以下に示します。

- 『*Document-Based App Programming Guide for Mac*』は、OS X用のドキュメントベースアプリケーションを開発する手順をまとめています。iOS用とOS X用のアプリケーション間でドキュメントを共有したい場合は、この資料を読んでおいてください。
- 『*Uniform Type Identifiers Overview*』およびその関連資料では、ドキュメントタイプの識別に用いる、UTI（Uniform Type Identifier）について説明しています。
- 『*File Metadata Programming Guide*』では、`NSMetadataQuery`およびその関連クラスを用いた検索の手順を説明しています。メタデータクエリを使って、iCloudに格納されたアプリケーションのドキュメントを検索することになります。

ドキュメントベースアプリケーションの設計

UIDocumentはiOS 5.0で導入されたクラスで、iOS用のドキュメントベースアプリケーションでは中心的な役割を果たします。これは抽象基底クラスなので、実際に役立てるためには、サブクラスを作成し、アプリケーションに合わせて調整する必要があります。iCloudストレージに統合されたモバイル環境で、効率的なドキュメント処理ができるよう、アプリケーション固有のコードを、UIDocumentのサブクラスに追加するのです。

ドキュメントベースアプリケーションの長所

あるアプリケーションのアイディアが浮かんでも、その設計に取りかかる段になると、さまざまな選択肢を評価しなければなりません。どのようなアプリケーションスタイルがよいでしょうか（「Master-Detail」型、ページベースのユーティリティ、OpenGLのゲームなど）。独自の描画処理を組み込む、ジェスチャやタッチイベントに応答する、画像や音声を取り込む、などの処理は必要でしょうか。どのようなデータモデルを採用するべきでしょうか（Core Dataを使うか、など）。アプリケーションをドキュメントベースにするか否か、さまざまな判断基準があって複雑に見えるかも知れませんが、要するに、ユーザがこのアプリケーションをどのように使うと期待するか、を見極めることです。

情報内容を開き、目で見ながら編集し、指定した名前で保存する、という使い方を期待するならば、ドキュメントベースアプリケーションは理想的、あるいは不可欠であると言ってもよいでしょう。ここでいう「情報の入れ物」、すなわちドキュメントは、ひとつひとつ内容が異なります。ドキュメントベースアプリケーションは、デスクトップシステムにもモバイルシステムにもさまざまな種類があり、もちろん、日頃から慣れ親しんでいることでしょう。すぐに思いつくものとして、ワードプロセッサ、スプレッドシート、描画プログラムなどがあります。iCloud技術と連携すれば、アプリケーションをドキュメントベースにする利点がより大きくなります。たとえばMac OS Xデスクトップシステム上のアプリケーションでドキュメントを作成した後、同期や複製の作業をしなくても、iPad上で動作するiOS用のアプリケーションで、同じドキュメントを編集できるのです。そのようなアプリケーションであれば、使ってみたいというユーザはますます増えることでしょう。

ドキュメントベースアプリケーションを開発する際、UIDocumentを利用すれば、最小限のプログラミング作業で各種機能を組み込むことができます。

- **iCloudストレージとの統合。**UIDocumentには、iCloudストレージとの間でドキュメントを読み書きする機能があります。この処理はNSFilePresenterプロトコルに基づくもので、NSFileCoordinatorクラス、NSFileManagerクラスの、適切なメソッドを呼び出して行います。

- **バックグラウンドでドキュメントデータを読み書き。**ドキュメントを同期的に読み書きすると、その間、アプリケーションが応答しなくなる可能性があります。`UIDocument`はこれを避けるため、バックグラウンドのディスパッチキューを用い、非同期に読み書きするようになっています。
- **「保存操作なし」モデル。**このモデルに基づくドキュメントベースアプリケーションでは、意識的にドキュメントを保存することがほとんどありません。`UIDocument`は、ユーザの作業状況に応じた時間間隔で、自動的にドキュメントデータを保存します。ドキュメントベースアプリケーションを「保存操作なし」にするためには、取り消し処理や状態変化の追跡処理を実装する必要があります（「状態変化の追跡、取り消し処理」（51 ページ）を参照）。
- **安全な保存処理。**`UIDocument`はドキュメントデータを安全に保存します。何らかの原因で保存処理が中断しても、ドキュメントデータの整合性が取れなくなることはありません。`UIDocument`ではこれを、ドキュメントの最新版を一時ファイルに書き出した後、現行ドキュメントを置き換える、という形で実装しています。
- **エラーや版の食い違いへの対処。**`UIDocument`は、ドキュメントの版同士の間に食い違いがあると、アプリケーションに通知するようになっています。アプリケーション側では、自らその食い違いを解決しても構いませんし、ユーザに選ばせることも可能です。`UIDocument`は、保存操作に失敗した場合も、同様に通知するようになっています。「ドキュメントのライフサイクル管理」（32 ページ）ではこの通知を監視する方法を説明します。「ドキュメントの版の食い違い解消」（54 ページ）では、版の食い違いを処理するための戦略を考察します。

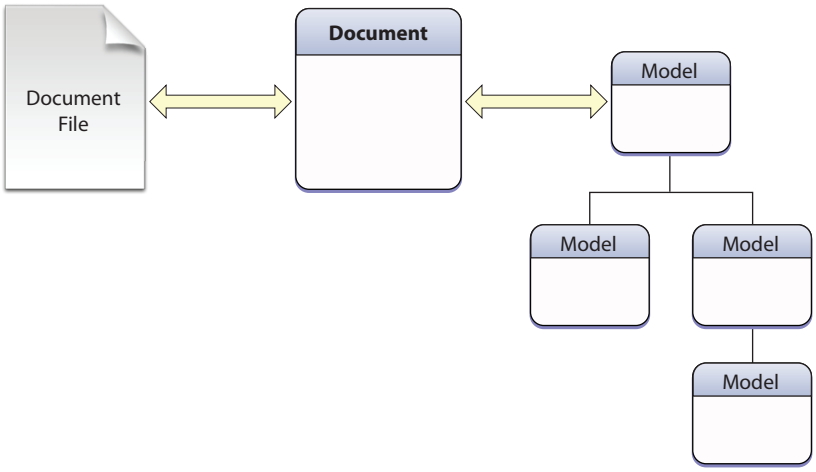
iOS上のドキュメント

「ドキュメント」は一般に「情報の入れ物」と定義されますが、この入れ物は立場によってさまざまな見方ができます。ユーザにとってドキュメントは、テキスト、画像、図形などの情報です。作成し、編集し、一意的な名前で保存することができます。あるいは、ドキュメントファイル、すなわち、ドキュメントデータを永続的なディスク上に表現したものを指すこともあります。さらに、「ドキュメント」という言葉で、`UIDocument`オブジェクトを表すこともあります。同じドキュメントデータをメモリ上に展開し、管理するオブジェクトです。

ドキュメントオブジェクトは、ディスク上の表現とメモリ中の表現を相互に変換する上でも、重要な役割を果たします。`UIDocument`クラスと連携して、ドキュメントデータをファイルに書き出し、あるいはファイルから読み込むのです。書き出しについては、ドキュメントが出力するのはデータのスナップショットで、これをそのままドキュメントファイルとして書き出すことになります。読み込み

に関しては、ドキュメントはデータを受け取り、それに基づいてモデルオブジェクトを初期化します。ドキュメントはある意味で、ファイルに格納されたデータとその内部表現を結ぶ、「導管」の役割を果たします。図 1-1にこの関係を示します。

図 1-1 ドキュメントファイル、ドキュメントオブジェクト、ドキュメントオブジェクトが管理するモデルオブジェクトの関係



iCloudストレージ上のドキュメント

iCloudでは、ファイルは、アプリケーションに関連づけられたコンテナディレクトリ内に、ローカルに置かれます。このディレクトリ以下は所定の構成になっており、最も重要なものとして`Documents`というサブディレクトリがあります。iCloudスキームでは、`Documents`サブディレクトリに書き出したファイルやファイルパッケージは、ドキュメントファイルと見なされます。ドキュメントベースアプリケーションで作成したものでなくても、これは変わりません。それ以外に書き出したファイルは、データファイルと見なされます。

ドキュメントオブジェクトは、`UIDocument`クラスが`NSFilePresenter` プロトコルに準拠しているので、ファイルプレゼンタでもあります。ファイルプレゼンタは`NSFileCoordinator`オブジェクトと併用して、アプリケーションのオブジェクト間、あるいはアプリケーションとほかのプロセスとの間で、ファイルやディレクトリへのアクセス処理を調整するために使います。また、ほかのクライアントが同じファイルやディレクトリにアクセスした場合にも、ファイルプレゼンタが関与します。

ユーザがiCloudストレージに対してドキュメントを保存するよう要求した場合、ドキュメントベースアプリケーションはドキュメントファイル（ファイルパッケージを含む）を、iCloudコンテナディレクトリの`Documents`サブディレクトリに格納しなければなりません。詳しくは「ドキュメントのライフサイクル管理」 （32 ページ）を参照してください。iCloudモバイルコンテナについては、『*iOS App Programming Guide*』の「iCloud Storage」を参照してください。

UIDocumentオブジェクトのプロパティ

ドキュメントオブジェクトにはいくつか、これを特徴づけるプロパティがあり、多くはドキュメントデータの管理者としての役割に関係しています。いずれもUIDocumentクラスで宣言されています。

- **ファイルURL。**ドキュメントには、ローカルファイルシステム内であれ、iCloudストレージ上であれ、それを格納する場所が必要です。この場所を表すのがfileURLプロパティです。UIDocumentオブジェクトを作成する際には、その初期化メソッドであるinitWithFileURL:に、引数としてファイルURLを指定しなければなりません。
- **ドキュメント名。**UIDocumentは、ファイルURLのうちファイル名に当たる部分をもとにドキュメント名の初期値を決め、localizedNameプロパティに設定します。このプロパティの取得メソッドをオーバーライドして、ローカライズされた独自のドキュメント名を返すようにすることもできます。

> **注意**　ドキュメントの表示名は、ファイル名と対応していなくても構いません。さらに、ドキュメント名を指定するよう、ユーザに要求するべきでもありません。この件について詳しくは、「新規ドキュメントの作成」（33 ページ）を参照してください。

- **ファイルタイプ。**ファイルタイプは、ファイルURLの拡張子から導かれるUTI（Uniform Type Identifier）で、fileTypeプロパティに設定します。詳しくは「iOSがアプリケーションのドキュメントを識別する方法」（21 ページ）を参照してください。
- **修正日。**ドキュメントファイルを最後に修正した日付。この値はfileModificationDateプロパティに保存します。版の食い違いを解決するなどの目的で使います。
- **ドキュメントの状態。**ドキュメントの状態はライフサイクルの経過に従って変遷します。たとえば、保存時にエラーが発生した、版の食い違いがある、などを表します。UIDocumentは現在のドキュメントの状態を、documentStateプロパティで管理します。状態変化を監視する方法について詳しくは、「ドキュメント状態の変化の監視とエラー処理」（46 ページ）を参照してください。

ドキュメントベースアプリケーションの設計における検討事項

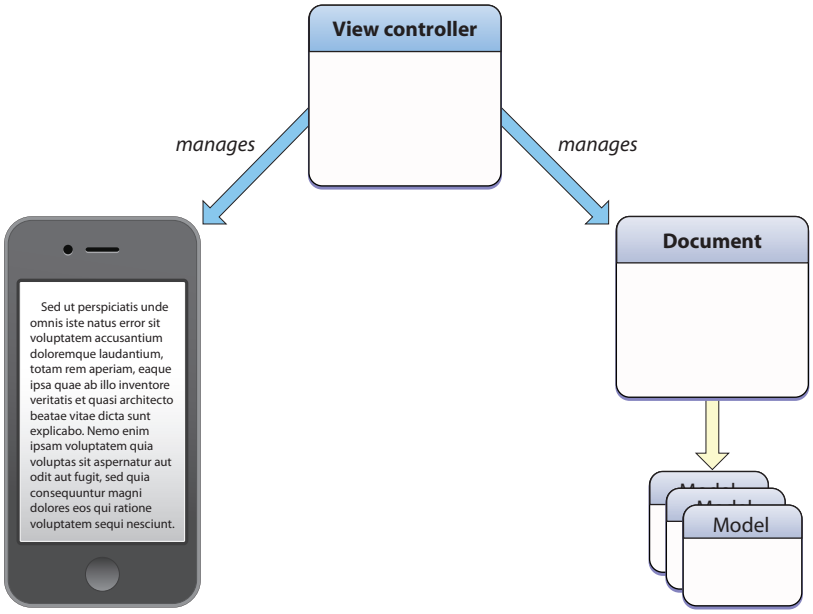
ドキュメントベースアプリケーションのコードを実際に書き始める前に、以下のような設計上の判断事項を検討しておいてください。

オブジェクト間の関係の定義

どのようなアプリケーションにも言えることですが、「モデル-ビュー-コントローラ」（MVC）デザインパターンに基づくドキュメントベースアプリケーションにおいても、オブジェクト全体の関係を検討しておくべきでしょう。ドキュメントについては次のようなMVC型の設計を推奨しますが、オブジェクトやその関係を独自に設計しても構いません。

MVCの用語では、ドキュメントはモデルコントローラであって、ドキュメントの内容を表すモデルオブジェクトを「所有」し管理します。ドキュメントオブジェクト自身は、ビューコントローラが所有し管理します。ビューコントローラは、その定義上当然ながら、ドキュメントオブジェクトが管理するドキュメントの内容を表示する、ビューも管理します。すなわち、このような関係で結ばれたオブジェクト間を仲介するコントローラであって、ビューに表示するために必要なデータをドキュメントオブジェクトから取得し、また、ユーザが入力または変更したデータをドキュメントオブジェクトに渡します。図 1-2に、これらのオブジェクトの関係を図示します。

図 1-2 ビューコントローラは、ドキュメントオブジェクトと、ドキュメントデータを表現するビューの、両方を管理する



ビューコントローラには、そのビューが埋め込まれているのと同様に、ドキュメントオブジェクトも宣言済みプロパティとして埋め込まれています。アプリケーションは、ビューコントローラのインスタンスを作成する際、ドキュメントオブジェクト、またはドキュメントのファイルURLを渡して初期化します（ビューコントローラはこの情報をもとにドキュメントオブジェクトを作成）。

もちろん、別のビューコントローラ（およびそのビュー）を持たせても、さらにはほかのモデルオブジェクトを使うようにしても構いません。どのような場合に別のビューコントローラが必要になるか、については「ユーザインターフェイスの設計」（15 ページ）を参照してください。

ユーザインターフェイスの設計

UIKitにもほかの開発ツールにも、ドキュメントベースアプリケーションのユーザインターフェイスに関する支援機能はありません。`UIDocumentView`、あるいは`UIDocumentViewController`といったクラスはないのです。しかしこれは逆に、競争力の高いユーザインターフェイスを開発する機会ととらえるべきでしょう。

とはいえ、ドキュメントベースアプリケーションである以上どうしても必要な機能がいくつかあり、そのためのユーザインターフェイス要素が欠かせません。たとえば次のような機能です。

- ドキュメントを表示、編集する
- 新規ドキュメントを作成する
- アプリケーションが管理するドキュメントリストからドキュメントを選択する
- 選択したドキュメントを開き、閉じ、削除する
- 選択したドキュメントをiCloudストレージに置く（およびiCloudストレージから削除する）
- エラー（版の食い違いを含む）の状況を表示する
- 変更を取り消し、やり直す（推奨するが必須ではない）

アプリケーション設計時には、こういった処理の実装に必要な、ビューコントローラ、ビュー、コントロールも検討してください。

ドキュメントデータのタイプ、形式、戦略の選択

ドキュメントベースアプリケーションのデータモデルを設計する際には、次のような事項の検討が重要です。

どのようなタイプのドキュメントか

ドキュメントにはドキュメントタイプを設定する必要があります。ドキュメントタイプとは、ドキュメントを特徴づけ、アプリケーションを対応づける、一連の情報のことです。ドキュメントタイプには、名前、アイコン（必須ではない）、ハンドラのランク、UTI（Uniform Type Identifier）が設定してあり、ファイル拡張子（複数でも可）と組にして扱います。同じドキュメントをiOSとMac OS Xの両方で編集できるようなアプリケーションの場合、ドキュメントタイプの情報が整合していなければなりません。

「プロジェクトの作成と設定」（20 ページ）では、iOS用のドキュメントベースアプリケーションで、ドキュメントタイプを指定する手順を説明しています。UTIについては『*Uniform Type Identifiers Overview*』、広く使われているUTIについては『*Uniform Type Identifiers Reference*』を参照してください。

ファイルに書き出すドキュメントデータをどのように表現するか

大きく分けて3つの選択肢があります。

- **Core Data（データベース）を利用。**Core Dataは、オブジェクトグラフを管理し永続化するための技術およびフレームワークです。データベースシステムとして、組み込みのSQLiteデータライブラリが使えます。`UIManagedDocument`は`UIDocument`のサブクラスで、Core Dataを利用するドキュメントベースアプリケーションを想定しています。
- **組み込みのオブジェクト形式。**`UIDocument`では、ドキュメントデータを表現するネイティブなタイプとして、`NSData`および`NSFileWrapper`オブジェクトが使えます。`NSData`は構造のない単独のファイル、`NSFileWrapper`はファイルパッケージ（実体はディレクトリであるがiOSは単一のファイルとして扱う）を想定しています。いずれかを`UIDocument`に渡せば、それだけでファイルシステムへの保存ができます。
- **独自のオブジェクト形式。**`NSData`でも`NSFileWrapper`でもないタイプでドキュメントデータをファイルに書き出す場合は、関係する`UIDocument`のメソッドをオーバーライドしてください。ただし、`UIDocument`に組み込まれているのと機能的には同じ処理コードを記述しなければならないこと、複雑になれば誤りも入り込みやすくなることを、頭に入れておかなければなりません。詳細については、『*UIDocument Class Reference*』を参照してください。

ドキュメントデータをファイルではなくデータベースに保存する必要があるか

ドキュメントオブジェクトで管理するデータセットが大規模なオブジェクトグラフを成し、しかしアプリケーションが一度に必要とするのはごく一部である場合、`UIManagedDocument`とCore Dataを利用するとよいでしょう。ドキュメントベースアプリケーションにCore Dataを取り入れれば、次のようなさまざまな利点があります。

- ドキュメントデータの増分読み込み、増分書き出し
- 取り消し/やり直し操作の支援
- 版の食い違いの自動解消支援
- プラットフォームをまたがるアプリケーションのデータ互換性

一方、Core Dataには、あまりにも複雑すぎる、という問題点もあります。その概念やクラス、技術を把握するには多少時間がかかります。さらに詳しくは、『*Core Data Programming Guide*』および『*UIManagedDocument Class Reference*』を参照してください。

組み込みのオブジェクト形式（NSData、NSFileWrapper）のうち、どちらがアプリケーションに適しているか

ドキュメントデータを管理するオブジェクトとして`NSData`と`NSFileWrapper`のどちらが適しているかは、データがどの位複雑か、ドキュメントのモデルオブジェクトグラフの一部を独立したファイルに書き出す必要があるか、によって決まります。たとえば、ドキュメントデータが単なるプレーンテ

キストであれば、NSDataがそのコンテナとして適しています。一方、たとえばテキストと画像のように*複数*のコンポーネントから成る場合は、NSFileWrapperのメソッドを使って、データを格納するファイルパッケージを作ってください。

ファイルラッパオブジェクト（およびこれが表すファイルパッケージ）には、単なるバイナリデータオブジェクトに比べ、次のような長所があります。

- 増分保存に対応。単独のバイナリデータオブジェクトと違い、ファイルラッパにドキュメントデータ（たとえばテキストと画像）を収容すれば、テキストを変更したとき、テキストを収容するファイルだけをディスクに書き出せばよいことになります。これには性能改善の効果があります。
- ファイルラッパ（およびファイルパッケージ）を使えば、ドキュメントの版管理が容易になります。ファイルパッケージに、たとえばプロパティリストファイルを入れて、版情報その他、ドキュメントに関するメタデータを保存しておけるからです。

「ドキュメントデータをファイルパッケージに格納」（30 ページ）では、NSFileWrapperオブジェクトを使って、ファイルパッケージに書き出せる形でドキュメントデータを表現する方法を解説し、例を示します。

モデルオブジェクトグラフをアーカイブし、NSDataオブジェクトでドキュメントファイルに書き出すことができるか

可能です。ただし、事前に考慮しておくべき事項があります。ドキュメントのモデルオブジェクトグラフのある部分を独立したファイルに書き出せるのであれば、その部分はアーカイブしないでください。代わりに、NSDataアーカイブと、分割した部分を、それぞれ独立したコンポーネントとしてファイルパッケージに保存します。

ドキュメントファイルが非常に大きくなった場合はどうするか

ドキュメントファイルに保存するデータが大きくなると、ユーザが使いやすいよう、増分読み込み/増分書き出しが必要になるかも知れません。方針がいくつか考えられます。

- UIManagedDocumentを使う方法。先にも述べたように、Core Dataには増分読み込み/増分書き出しの機能があります。
- ドキュメントデータを個別のコンポーネントに分け、ファイルパッケージとして保存する方法。
- UIDocumentの低レベルメソッドをオーバーライドし、増分読み込み/増分書き出しのコードを独自に記述する方法（詳しくは『*UIDocument Class Reference*』を参照）。データタイプに応じて、増分読み込み/増分書き出し用の適切なAPIを使うようにしてください。たとえばAV Foundationフレームワークには、大規模マルチメディアファイル用の増分読み込みメソッドがあります。

ドキュメントをどちらのプラットフォーム上でも編集できるようにするためには、何を考慮すればよいか

iOSモバイルデバイス（iPhone、iPad、iPad touch）とMac OS Xデスクトップシステムの両方でドキュメントを作成、編集できれば、アプリケーションの競争力が向上するでしょう。しかしそのためには、ドキュメントデータの形式を、どちらのプラットフォームでも扱えるようにしなければなりません。ドキュメントデータの互換性に関して、検討を要する重要な事項を以下に示します。

- 一方のプラットフォームでしか使えない技術がいくつかあります。たとえば、Mac OS X上ではドキュメント形式としてRTFが使えますが、iOSの場合、テキストシステムがリッチテキストに対応していません。
- 各プラットフォームの対応するクラスに互換性がない場合があります。特に問題になるのは、色（`UIColor`と`NSColor`）、画像（`UIImage`と`NSImage`）、ベジエパス（`UIBezierPath`と`NSBezierPath`）でしょう。たとえば`NSColor`オブジェクトは色空間（`NSColorSpace`）の概念に基づき定義されていますが、UIKitには色空間のクラスがありません。

  一方のクラスでドキュメントプロパティを定義した場合、（ドキュメントデータを書き出す際に）もう一方で正確に再現できる形式に変換する手段を考えなければなりません。ひとつの方法は、どちらのプラットフォームでも対応できるよう、機能の低い方のフレームワークに合わせて「降格」させることです。たとえば`UIColor`には`CIColor`プロパティがあって、色を表すCore Imageオブジェクトを保持するようになっています。Mac OS X側では`NSColor`オブジェクトを、`CIColor`オブジェクトから、`colorWithCIColor:`というクラスメソッドで生成できます。
- 既定の座標系がプラットフォームによって異なるため、描画方法に影響があります（この件については『*Drawing and Printing Guide for iOS*』の「Default Coordinate Systems and Drawing in iOS」 in *Drawing and Printing Guide for iOS*を参照）。
- ドキュメントのモデルオブジェクトグラフ（一部または全部）をアーカイブする場合、モデルオブジェクトをエンコード/デコードする際に、`NSCoder`のメソッドで、プラットフォームに応じた変換をする必要があります。

ドキュメントベースアプリケーションの概要

ドキュメントベースアプリケーションを開発するからと言って、文章ベースでない場合に比べ、作業量が大幅に増えるということはほとんどありません。主な違いは、`UIDocument`のサブクラスを独自に作成し、ドキュメントをそのライフサイクルを通して管理しなければならないこと（iCloudストレージへの統合を含む）です。この章では、ほとんどのアプリケーションに必要となるであろう、ドキュメント特有の処理を概観し、iOS上でドキュメントベースアプリケーションの開発プロジェクトを作成、設定する一般的な手順を解説します。

ドキュメントベースアプリケーションの開発に必要な作業

ドキュメントベースアプリケーションを開発するためには、次のような作業が必要です。

- `UIDocument`のサブクラスを独自に作成すること。ドキュメントデータのスナップショットを作る処理、ドキュメントファイルの内容をもとにドキュメントのモデルオブジェクトを初期化する処理を実装します。

  「独自のドキュメントオブジェクトの作成」（25 ページ）で、オーバーライドが必要なメソッドと、`NSFileWrapper`オブジェクトでドキュメントデータを管理する方法（ドキュメントデータをファイルパッケージに格納する場合のみ）を説明しています。

- ユーザが新規ドキュメントを作成し、既存のドキュメントを選択し、開けるようにすること。また、反対に、ドキュメントを閉じ、削除する処理も実装します。

  ドキュメントを作成し、または開いた後は、その内容をビューに表示しなければならないことに注意してください。

  この処理の前提となる要件と具体的な実装方法について、「新規ドキュメントの作成」（33 ページ）、「ドキュメントを開く/閉じる」（36 ページ）、「ドキュメントの削除」（48 ページ）で説明しています。さらに、ドキュメントデータの表示を管理する例も載っています。

- 取り消し管理や状態変化の追跡処理を実装し、ドキュメントデータを自動保存できるようにすること（「保存操作なし」モデル）。

  詳しくは「状態変化の追跡、取り消し処理」（51 ページ）を参照してください。

- ドキュメント（通常はユーザが選択したもの）をiCloudストレージに移すこと。また、ユーザが指示したときに、iCloudストレージからドキュメントを削除すること。

  実際の手順については「ドキュメントをiCloudストレージに、またはiCloudストレージから移動」（42 ページ）を参照してください。

- ドキュメントの状態が変化した旨の通知を監視し、エラーが発生した場合は適切に対応すること。

  その一般的な手順については「ドキュメント状態の変化の監視とエラー処理」（46 ページ）を参照してください。
- ドキュメントの版に食い違いが生じた場合、ユーザに通知し、解決する手段を与えること。

  ユーザに版の食い違いを通知する方法、解決する方法については、「ドキュメントの版の食い違い解消」（54 ページ）を参照してください。

そのほか、ドキュメントの印刷、綴りチェック、電子メール送信など、さまざまな機能を付け加えることが可能です。

大容量ドキュメントファイルの増分読み込み/書き出し、組み込み以外のドキュメントデータ形式の処理など、さらに高度な要件がある場合は、『*UIDocument Class Reference*』を参照してください。こういった機能を追加する場合、`UIDocument`のメソッドをオーバーライドすることになります。

プロジェクトの作成と設定

ドキュメントベースアプリケーションを開発するXcodeプロジェクトを作成する際には、適切なテンプレートを選択します（ドキュメントベースアプリケーションに特有のテンプレートではないことに注意）。通常、アプリケーションを起動して最初に表示されるのは、既存のドキュメントを選択する、または新規に作成するための画面でしょう。この目的にはXcodeの「Master-Detail」テンプレートが適しているので、今後出てくるコード例ではこれを使います。このテンプレートには、テーブルビュー（iPhone用）または分割ビュー（iPad用）による初期画面があります。プロジェクトではストーリーボードも使うことになるので、忘れずに選択してください。

> **注意** Core Dataを使ってドキュメントデータを管理する場合は、「New Project」アシスタントで、忘れずに「Use Core Data」を選択してください。

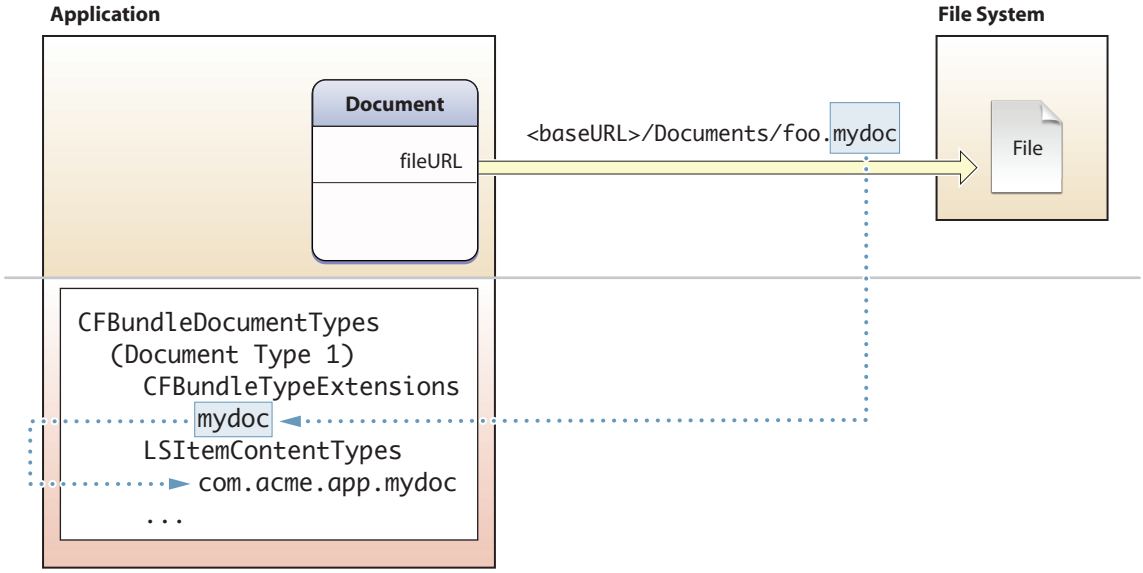
アプリケーションの設計に基づき（「ドキュメントベースアプリケーションの設計」（10 ページ）を参照）、アプリケーションに必要となる、ビューコントローラのサブクラスを生成します。この時点で必要なのは、最小限の宣言を記述したヘッダファイルとソースファイルだけです。次に、アプリケーションのユーザインターフェイスを、プロジェクトストーリーボードで作成します（汎用的なアプリケーションであれば複数のストーリーボードを使用）。カスタムビューコントローラに、ストーリーボードのビューコントローラプレースホルダを対応させてください。

続いて、Xcodeの対象設定画面で、アプリケーションが扱うドキュメントのタイプを指定し、プロジェクトの設定をします。

iOSがアプリケーションのドキュメントを識別する方法

iOSにおいて、ドキュメントオブジェクトの属性として最も重要なのはファイルURL（`fileURL`）です。その目的のひとつに、iOSに対して、当該ドキュメントの形式を認識できるアプリケーションを知らせる、という役割があります。ファイルURLの末尾は拡張子（たとえば`html`）になっており、これがUTI（Uniform Type Identifier）に合致します（たとえば`public.html`のように）。UTI（Uniform Type Identifier）はドキュメントタイプの識別子として重要です。`UIDocument`は拡張子を手がかりにドキュメントタイプUTIを調べ（図2-1を参照）、`fileType`プロパティに設定します。OS X用のドキュメントベースアプリケーションと違い、iOSの場合、`UIDocument`のサブクラスにドキュメントタイプを関連づける必要はありません。

図 2-1　iOSはファイルURLの拡張子を手がかりにドキュメントのUTIを調べる



ドキュメントタイプUTIはシステムで定義できます。広く使われている識別子のリストは、『*Uniform Type Identifiers Reference*』の「System-Declared Uniform Type Identifiers」を参照してください。ドキュメントベースアプリケーションは、独自の（独占的な）UTIを定義することも可能で、実際にしばしば行われています。独自のUTIを宣言する場合はさらに、当該UTIをエクスポートして、オペレーティングシステムが認識できるようにしなければなりません。

ドキュメントタイプの宣言

Xcodeでドキュメントタイプを宣言するためには、対象の「Info」設定画面の「Add」ボタンを押し、ポップアップメニューから「Add Document Type」を実行してください。「Untitled」という表示の脇にある三角形をクリックするとプロパティ欄が現れ、表 2-1に示すプロパティを設定できます。

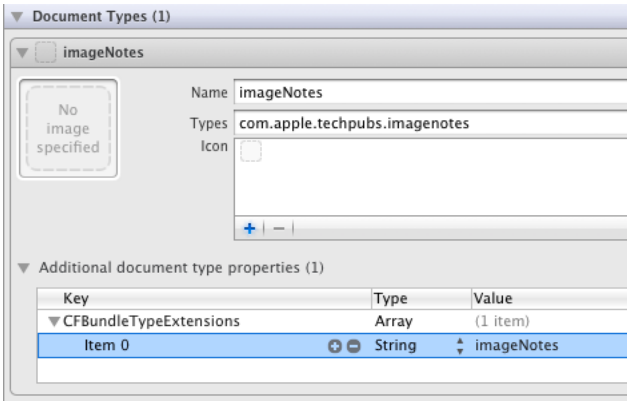
表 2-1 ドキュメントタイプを定義するためのプロパティ（`CFBundleDocumentTypes`）

キー	Xcodeのフィールド	値と説明
`LSItemContentTypes`	Types	UTI文字列の配列。通常はドキュメントタイプごとに1つだけ指定。
`CFBundleTypeName`	Name	ドキュメントタイプを表す名前（必要に応じて指定）。
`CFBundleType-IconFiles`	Icon	アプリケーションバンドル内のアイコン画像ファイルパスの配列。
`CFBundleType-Extensions`	「その他のドキュメントタイププロパティ」テーブル内。	ドキュメントUTIと組にするファイル名拡張子の配列
`LSHandlerRank`	「その他のドキュメントタイププロパティ」テーブル内。	`Owner`、`Alternate`、`None`のいずれか（通常は`Owner`）。
`LSTypeIsPackage`	「その他のドキュメントタイププロパティ」テーブル内。	ドキュメントデータをファイルパッケージに格納する場合は`YES`。それ以外の場合は省略。

上記のキーについて詳しくは、『*Information Property List Key Reference*』の「CFBundleDocumentTypes」in *Information Property List Key Reference* を参照してください。

ドキュメントタイプを表すプロパティをすべて入力すると、Xcodeの「Document Types」領域は、図2-2のようになっているはずです。

図 2-2 Xcodeにおけるドキュメントタイプの指定



同じアプリケーションに、複数のドキュメントタイプを設定することも可能です。たとえばワードプロセッサの場合、通常ドキュメントを表すタイプと、整形済みドキュメントを表すタイプの両方を設定する、といった具合です。それぞれのタイプごとに、上記の作業をしてください。

ドキュメントUTIのエクスポート

ドキュメントを表す独自のUTIを定義した場合は、それをエクスポートする必要があります。Xcode上でエクスポートする場合は、対象の「Info」設定画面の「Add」ボタンを押し、ポップアップメニューから「Add Exported UTI」を実行してください。「Untitled」という表示の脇にある三角形をクリックするとプロパティ欄が現れ、表 2-2に示すプロパティを設定できます。

> **注意** UTIを宣言、エクスポートする手順については、『*Uniform Type Identifiers Overview*』の「Declaring New Uniform Type Identifiers」を参照してください。

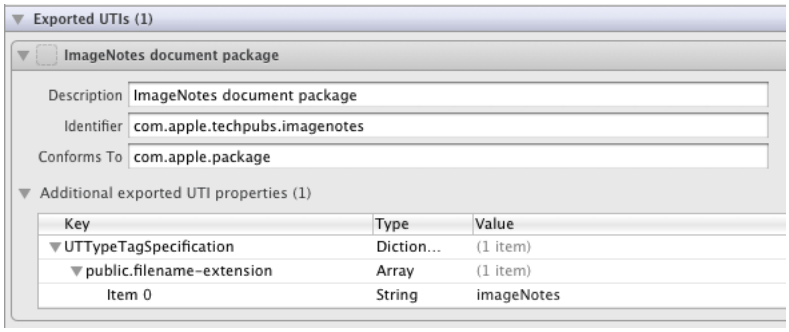
表 2-2 ドキュメントUTIをエクスポートするためのプロパティ（`UTExportedTypeDeclarations`）

キー	Xcodeのフィールド	値と説明
`UITypeIdentifier`	Identifier	独自のドキュメントUTI（文字列）。
`UTTypeConformsTo`	Conforms to	独自のドキュメントUTIが適合するUTI。データをファイルパッケージに格納する場合は`com.apple.package`を指定。
`UTTypeDescription`	説明	エクスポートしたタイプの説明（必要に応じて指定）。
`UTTypeTag-Specification`	「その他のエクスポート済みUTIプロパティ」テーブル内。	`public.filename-extension`という名前の配列を作成し、ドキュメントファイルの拡張子をすべて、項目として追加。

上記のキーについて詳しくは、『*Information Property List Key Reference*』の「CFBundleDocumentTypes」in *Information Property List Key Reference*を参照してください。

ドキュメントタイプを表すプロパティをすべて入力すると、Xcodeの「Exported UTIs」領域は、図 2-3 のようになっているはずです。

図 2-3 Xcodeにおける独自のドキュメントUTIのエクスポート



独自のドキュメントオブジェクトの作成

ドキュメントベースアプリケーションには、ドキュメントデータを表現し管理するために、UIDocumentのサブクラスのインスタンスが必要です。この章では、大部分のアプリケーションでオーバーライドする必要があるメソッドについて述べ、それ以外のメソッドのオーバーライドについても概説します。特にオーバーライドが必要なメソッド、すなわちloadFromContents:ofType:error:とcontentsForType:error:については、NSDataとNSFileWrapperの両方について、ドキュメントファイルを読み書きする例を示します。「ドキュメントデータをファイルパッケージに格納」（30 ページ）ではさらに、ドキュメントデータをファイルラッパオブジェクトで管理する方法を解説します。

この章で説明する以外にも、UIDocumentの適当なメソッドをオーバーライドすることにより、特別なやり方でドキュメントデータを読み書きすることができます（増分読み込み/増分書き出しなど）。しかし、そのためには複雑な要件を満たす必要があるので、できれば避ける方がよいでしょう。このようなオーバーライドについては『*UIDocument Class Reference*』を参照してください。

ドキュメントクラスのインターフェイスの宣言

Xcodeで、プロジェクトにObjective-Cのソースファイルとヘッダファイルを追加し、適切な名前をつけてください（名前に「Document」という文字列を含めておくと分かりやすいでしょう）。ヘッダファイルで、スーパークラスをUIDocumentに変更し、ドキュメントデータを保持するプロパティを追加します。リスト 3-1の場合、ドキュメントデータはプレーンテキストなので、これを保持するために必要なのはNSString型のプロパティがひとつだけです（このテキストをNSDataオブジェクトに変換し、ドキュメントファイルに書き出します）。

リスト 3-1　ドキュメントサブクラスの宣言（NSData）

```
@interface MyDocument : UIDocument {
}
@property(nonatomic, strong) NSString *documentText;
@end
```

リスト 3-2に、データ表現型としてNSFileWrapperオブジェクトを使う、別のアプリケーションの宣言例を示します（この章では、2つのアプリケーションのコード例を交互に示します）。ファイルラッパオブジェクトを保持するプロパティに加え、ファイルパッケージに入れる、テキストや画像コンポーネントを保持するプロパティもあります。

リスト 3-2　ドキュメントサブクラスの宣言（NSFileWrapper）

```
@interface ImageNotesDocument : UIDocument

@property (nonatomic, strong) NSString* text;
@property (nonatomic, strong) UIImage* image;
@property (nonatomic, strong) NSFileWrapper *fileWrapper;

@property (nonatomic, weak) id <ImageNotesDocumentDelegate> delegate;
@end

@protocol ImageNotesDocumentDelegate <NSObject>
-(void)noteDocumentContentsUpdated:(ImageNotesDocument*)noteDocument;
@end
```

このコードには、デリゲートと、これが従うプロトコルの宣言も追加されています。ドキュメントオブジェクトのビューコントローラは、自分自身をドキュメントオブジェクトのデリゲートにし、所定のプロトコルに従うよう設定することにより、ドキュメントファイルが更新されたとき、（noteDocumentContentsUpdated:メッセージを介して）通知を受け取ることができます。リスト 3-4 （27 ページ）に、noteDocumentContentsUpdated:メッセージが送られるタイミングと方法を示します。

ドキュメントデータの読み込み

アプリケーションが（ユーザの要求に応じて）ドキュメントを開くと、UIDocumentはドキュメントファイルの内容を読み込み、ドキュメントデータをカプセル化したオブジェクトを渡してloadFromContents:ofType:error:メソッドを呼び出します。このオブジェクトは、NSDataでもNSFileWrapperでも構いません。オーバーライドしたメソッド内では、ドキュメントの内部データ構造（すなわちモデルオブジェクト）を、渡されたオブジェクトの内容をもとに初期化します。

リスト 3-3の例では、渡されたNSDataオブジェクトから文字列を生成し、documentTextプロパティに代入しています。また、プロトコルメソッドを起動することにより、デリゲート（この場合はドキュメントのビューコントローラ）に更新されたドキュメント内容を通知しています。デリゲートメッセージを用いて通知しているのは、ドキュメントを開いたときだけではなく、iCloud上のドキュメントの更新/復元操作（revertToContentsOfURL:completionHandler:）によっても、このloadFromContents:ofType:error:メソッドが呼び出されるからです。

リスト 3-3　ドキュメントデータの読み込み（NSData）

```
- (BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName error:(NSError
**)outError {
    if ([contents length] > 0) {
        self.documentText = [[NSString alloc] initWithData:(NSData *)contents
encoding:NSUTF8StringEncoding];
    } else {
        self.documentText = @"";
    }
    if ([_delegate respondsToSelector:@selector(noteDocumentContentsUpdated:)]) {
        [_delegate noteDocumentContentsUpdated:self];
    }
    return YES;
}
```

複数のドキュメントタイプがある場合は、typeNameパラメータを調べて処理を振り分けます。ドキュメントタイプによって、ドキュメントデータオブジェクトの処理方法が異なるかも知れません。エラーのためドキュメントデータを読み込めない場合はNOを返します。エラーの詳細を記述した、NSErrorオブジェクトの参照を返してもよいでしょう。

リスト 3-4の例では、NSFileWrapperオブジェクト形式のドキュメントデータを扱っています。ここでは、単にオブジェクトをプロパティに代入しているだけです。

リスト 3-4　ドキュメントデータの読み込み（NSFileWrapper）

```
-(BOOL)loadFromContents:(id)contents ofType:(NSString *)typeName error:(NSError
**)outError {
    self.fileWrapper = (NSFileWrapper *)contents;
    if ([_delegate respondsToSelector:@selector(noteDocumentContentsUpdated:)]) {
        [_delegate noteDocumentContentsUpdated:self];
    }
    return YES;
}
```

このコード例では、ファイルラッパのテキストおよび画像コンポーネントを取り出し、該当するプロパティに代入する、という処理をしていません。textおよびimageプロパティの取得メソッドで、必要になった時点でこの処理をするものと想定しています。

ドキュメントデータのスナップショット作成

ドキュメントを閉じる、あるいは自動保存する際、UIDocumentはドキュメントオブジェクトにcontentsForType:error:メッセージを送ります。このメソッドをオーバーライドして、ドキュメントデータのスナップショットをUIDocumentに返して、実際にドキュメントファイルに書き出す処理を委ねます。リスト3-5では、ドキュメントデータのスナップショットを、NSDataオブジェクトの形で返しています。

リスト 3-5　ドキュメントデータのスナップショット作成（NSData）

```
- (id)contentsForType:(NSString *)typeName error:(NSError **)outError {
    if (!self.documentText) {
        self.documentText = @"";
    }
    NSData *docData = [self.documentText dataUsingEncoding:NSUTF8StringEncoding
allowLossyConversion:NO];
    return docData;
}
```

documentTextプロパティにまだ文字列値が代入されていなければ、空文字列を与えてから、NSDataオブジェクトを生成しています。

リスト3-6に、NSFileWrapperオブジェクトを返す形で、同じメソッドを実装した例を示します。最上位（ディレクトリ）のファイルラッパオブジェクトが存在しなければ、ここで作成します。これに含まれる2つのファイルラッパオブジェクト（通常ファイル）が存在しなければ、textプロパティ、imageプロパティの値をもとに作成します。その後、最上位のファイルラッパをUIDocumentに返し、ファイルシステム上にファイルパッケージを作成する処理を委ねます。ファイルパッケージとドキュメントについて詳しくは、「ドキュメントデータをファイルパッケージに格納」（30 ページ）を参照してください。

リスト 3-6　ドキュメントデータのスナップショット作成（NSFileWrapper）

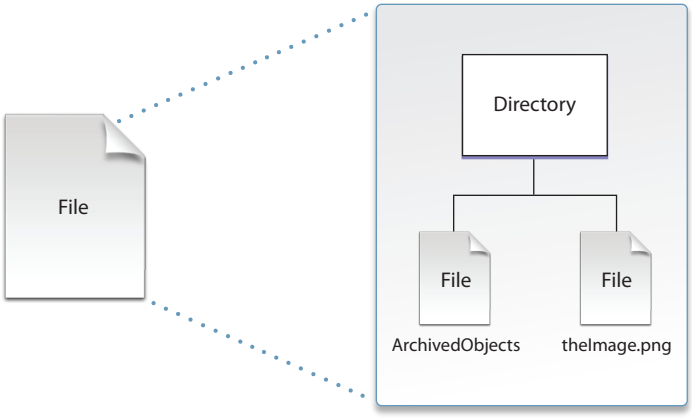
```
-(id)contentsForType:(NSString *)typeName error:(NSError **)outError {

    if (self.fileWrapper == nil) {
       self.fileWrapper = [[NSFileWrapper alloc] initDirectoryWithFileWrappers:nil];
    }
    NSDictionary *fileWrappers = [self.fileWrapper fileWrappers];
```

```
    if (([fileWrappers objectForKey:TextFileName] == nil) && (self.text != nil))
{
        NSData *textData = [self.text dataUsingEncoding:TextFileEncoding];
        NSFileWrapper *textFileWrapper = [[NSFileWrapper alloc]
initRegularFileWithContents:textData];
        [textFileWrapper setPreferredFilename:TextFileName];
        [self.fileWrapper addFileWrapper:textFileWrapper];
    }
    if (([fileWrappers objectForKey:ImageFileName] == nil) && (self.image != nil))
 {
        @autoreleasepool {
            NSData *imageData = UIImagePNGRepresentation(self.image);
            NSFileWrapper *imageFileWrapper = [[NSFileWrapper alloc]
initRegularFileWithContents:imageData];
            [imageFileWrapper setPreferredFilename:ImageFileName];
            [self.fileWrapper addFileWrapper:imageFileWrapper];
        }
    }
    return  self.fileWrapper;
}
```

ドキュメントデータをファイルパッケージに格納

ファイルパッケージには内部構造があり、これがNSFileWrapperクラスのメソッドに反映されています。ファイルラッパはファイルシステムのノード（ディレクトリ、通常ファイル、シンボリックリンクのいずれか）の実行時表現です。図 3-1に示すように、ファイルパッケージはファイルシステムのノード（通常はディレクトリ）とその中身を組にしたもので、オペレーティングシステムはこれを、構造を隠し、単一の実体として扱います。「バンドル」に似た考え方とも言えるでしょう。

図 3-1　ファイルパッケージの構造



プログラムでは、最上位ディレクトリに対応するファイルラッパを作成し、その内容である通常ファイルやサブディレクトリ（いずれも別のNSFileWrapperオブジェクトで表される）を追加していくことにより、ファイルパッケージを組み立てます。最上位ディレクトリ内のファイルラッパには適当な名前を関連づけます。

以上の概要を頭に入れた上で、次のコードを見てください。これはリスト 3-6 （28 ページ）に示したcontentsForType:error:メソッドの一部です。このメソッドで作成されるファイルパッケージは、テキストファイルと画像ファイルの2つから成ります（画像ファイルラッパの作成処理は省略）。

```
    if (self.fileWrapper == nil) {
        self.fileWrapper = [[NSFileWrapper alloc] initDirectoryWithFileWrappers:nil];
    }
    NSDictionary *fileWrappers = [self.fileWrapper fileWrappers];
    if (([fileWrappers objectForKey:TextFileName] == nil) && (self.text != nil))
{
        NSData *textData = [self.text dataUsingEncoding:TextFileEncoding];
        NSFileWrapper *textFileWrapper = [[NSFileWrapper alloc]
initRegularFileWithContents:textData];
        [textFileWrapper setPreferredFilename:TextFileName];
```

```
        [self.fileWrapper addFileWrapper:textFileWrapper];
    }
```

まず、最上位ディレクトリがなければ作成します。次に、テキストファイルに対応するファイルラッパがなければ、textプロパティの内容（文字列）をもとに作成します。このファイルラッパに適当なファイル名を与え、最上位ディレクトリファイルラッパに追加していきます。

NSFileWrapperについて詳しくは、『*NSFileWrapper Class Reference*』を参照してください。また、ドキュメントファイルパッケージに必要なInfo.plistプロパティについては、「ドキュメントUTIのエクスポート」（23 ページ）を参照してください。

必要に応じてオーバーライドするその他のメソッド

UIDocumentのメソッドにはほかにも、多くのドキュメントベースアプリケーションでオーバーライドするであろうものがあります。

- disableEditing、enableEditing - UIDocumentは、ドキュメント内容をユーザが安全に変更できない状況になるとdisableEditingを呼び出します。iCloud上に新しい版があるとき、あるいは取り消し処理中などがこれに当たります。この間、編集できないようにしたい場合は、このメソッドを適切に実装してください。その後、再び安全に編集できるようになると、UIDocumentはenableEditingを呼び出します。

  > **注意** 代替として、ドキュメントの状態変化を表す通知を監視し、状態がUIDocumentStateEditingDisabledになったとき、別の状態に変わるまで編集できないようにする、という方法もあります。この件について詳しくは、「ドキュメント状態の変化の監視とエラー処理」（46 ページ）を参照してください。

- savingFileType — デフォルトでは、fileTypeプロパティの値を返すメソッドです。何らかの理由で、別のファイルタイプで保存したい場合は、該当するファイルタイプUTIを返すようオーバーライドしてください。画像をRTFファイルに追加した場合に、RTFDファイルパッケージとして保存しなければならない、という例があります（Mac OS X）。

ドキュメントのライフサイクル管理

ドキュメントは作成から削除に至るライフサイクルをたどります。ドキュメントベースアプリケーションは、このサイクルに沿ってドキュメントを管理しなければなりません。以下に列挙するような、ライフサイクルに関わるイベントの多くは、ユーザがそのきっかけを作ります。

- ユーザがまずドキュメントを作成する。
- ユーザは既存のドキュメントを開き、アプリケーションはドキュメントのビューにその内容を表示する。
- ユーザがドキュメントを編集する。
- ユーザが、ドキュメントをiCloudストレージに置くよう指示し、あるいはiCloudストレージがら削除するよう指示する。
- 編集中に保存その他の操作をし、あるいはエラーや版の食い違いが発生する。アプリケーションはその状況を調べ、自ら解決するか、ユーザに通知する。
- ユーザが選択したドキュメントを閉じる。
- ユーザが既存のドキュメントを削除する。

以下の各節では、上記の各操作に応じ、ドキュメントベースアプリケーションが実行するべき処理について解説します。

ドキュメントファイルの保存場所の設定

アプリケーションが扱うドキュメントはすべて、ローカルサンドボックスまたはiCloudのコンテナディレクトリに保存されます。iCloudのどこに保存するか、ドキュメントごとにいちいち選択しなければならないようであっては困ります。

あるデバイス上で最初にアプリケーションが起動された時点で、次のような処理をしてください。

- iCloudの設定がまだであれば、設定するかどうかユーザに訊ねます（する場合は「Launch Settings」画面に切り替えるとよいでしょう）。
- iCloudが設定済みであっても、このアプリケーションから使えるようになっていなければ、使えるようにするかどうか（すなわち、ドキュメントをすべてiCloudに保存するかどうか）を訊ねます。得られた応答は、環境設定として保存してください。

アプリケーションはこの環境設定に基づき、ローカルサンドボックス、またはiCloudコンテナディレクトリに、ドキュメントファイルを書き出します（詳しくは「ドキュメントをiCloudストレージに、またはiCloudストレージから移動」（42 ページ）を参照）。設定内容は「Settings」アプリケーションに表示して、ユーザがいつでも変更できるようにしなければなりません。

新規ドキュメントの作成

ドキュメントオブジェクト（`UIDocument`のサブクラスのインスタンス）にはファイルURLを与える必要があります。これはドキュメントファイルの保存場所を表すもので、環境設定に応じ、ローカルサンドボックス内、またはiCloudコンテナディレクトリにあります。さらに、新規ドキュメントには名前を与えることができます。以下、ファイルURL、ドキュメント名、新規ドキュメントの作成について、そのガイドラインと手順を解説します。

ドキュメントファイル名とドキュメント名の関係

`UIDocument`クラスでは、ドキュメントのファイル名とドキュメント名（あるいは表示名）に、対応関係があると想定しています。デフォルトでは、`UIDocument`はファイル名を、`localizedName`プロパティの値として保存しています。しかしアプリケーションは、ユーザが新規ドキュメントを作成する際、必ずドキュメントファイル名や表示名を決めるよう求めるべきではありません。

新規ドキュメントのファイル名を自動生成する、何らかの規約を決めておくべきでしょう。いくつか考えられる方針を示します。

- ドキュメントごとにUUID（汎用一意識別子、Universally Unique IDentifier）を生成する（アプリケーション固有の接頭辞をつけても可）。
- ドキュメントごとにタイムスタンプ（日付と時刻）を生成する（アプリケーション固有の接頭辞をつけても可）。
- 一連番号をつける（「Notes 1」、「Notes 2」などのように）。

ドキュメント名（表示名）に関しては、ドキュメントファイル名に意味があるならば（たとえば「Notes 1」のように）、初期値としてそれを使うのもよいでしょう。あるいは、ドキュメント中にテキストがあり、ユーザがドキュメントに何らかのテキストを入力するならば、その1行目（あるいはその一部）を表示名にすることも考えられます。作成後にドキュメント名を変更する、何らかの手段を用意しても構いません。

ファイルURLの組み立て、ドキュメントファイルの保存

適切なファイルURLがなければ、ドキュメントオブジェクトは作成できません。ファイルURLは、（環境設定で決められたドキュメント保存場所における）Documentsディレクトリのパス、ドキュメントファイル名、ドキュメントファイル拡張子という、3つの部分から成ります。ローカルサンドボックスにおける、Documentsディレクトリのパスを表すURLは、「ドキュメントファイル名とドキュメント名の関係」に示したような方法で取得できます。

リスト 4-1　ローカルサンドボックスにおける、アプリケーションのDocumentsディレクトリのURLを取得

```
-(NSURL*)localDocumentsDirectoryURL {
    static NSURL *localDocumentsDirectoryURL = nil;
    if (localDocumentsDirectoryURL == nil) {
        NSString *documentsDirectoryPath = [NSSearchPathForDirectoriesInDomains(
NSDocumentDirectory,
            NSUserDomainMask, YES ) objectAtIndex:0];
       localDocumentsDirectoryURL = [NSURL fileURLWithPath:documentsDirectoryPath];
    }
    return localDocumentsDirectoryURL;
}
```

ファイル拡張子は、ドキュメントタイプに応じて指定したものでなければなりません（「プロジェクトの作成と設定」（20 ページ）を参照）。拡張子を表すグローバル文字列を宣言しておくとよいでしょう。次に例を示します。

```
static NSString *FileExtension = @"imageNotes";
```

ファイルURLの最後は「ファイル名」に当たる部分です。「ドキュメントファイル名とドキュメント名の関係」（33 ページ）で説明したように、アプリケーションは最初、アプリケーションにとって意味のある何らかの規約に従い、ドキュメントファイル名を生成します。このファイル名をドキュメント名として使っても、あるいはドキュメントの1行目（またはその一部）をドキュメント名としても構いません。アプリケーションに、ドキュメントオブジェクト作成後、ユーザがドキュメント名を変更する機能を組み込んでもよいでしょう。

基底URL、ドキュメントファイル名、ファイル拡張子を連結した後、UIDocumentのサブクラスのインスタンスを作成し、このファイルURLを渡してinitWithFileURL:メソッドを起動することにより、初期化ができます。新規ドキュメントを作成する最後の処理は、所定のドキュメント保存場所に保存

することです（この時点では中身がなくても、いったん保存します）。「ドキュメントファイルの保存場所の設定」のように、これはドキュメントオブジェクトの`saveToURL:forSaveOperation:completionHandler:`メソッドを呼び出して行います。

リスト 4-2　新規ドキュメントをファイルシステムに保存

```
-(void)viewWillAppear:(BOOL)animated {
    [super viewWillAppear:animated];
    if (_createFile) {
        [self.document saveToURL:self.document.fileURL
            forSaveOperation:UIDocumentSaveForCreating completionHandler:^(BOOL
success) {
            if (success)
                _textView.text = self.document.text;
        }];
        _createFile = NO;
    }
    // .....
}
```

メソッドには、保存操作の種類を表すパラメータとして、`UIDocumentSaveForCreating`を渡します。最後のパラメータは完了ハンドラ、すなわち、保存終了時に呼び出されるコードブロックです。このブロックに渡されるパラメータは、保存操作が成功したか否かを表します。成功した場合、このコードはドキュメントテキストを、ドキュメント内容を表示するテキストビューの`text`プロパティに代入します。

注意 新規ドキュメントをアプリケーションのiCloudコンテナディレクトリに保存したい場合は、いったんローカルに保存した上で、`NSFileManager`の`setUbiquitous:itemAtURL:destinationURL:error:`メソッドでドキュメントファイルをiCloudストレージに移動する、という手順を推奨します（この呼び出しを`saveToURL:forSaveOperation:completionHandler:`メソッドの完了ハンドラに記述する方法も可）。詳しくは「ドキュメントをiCloudストレージに、またはiCloudストレージから移動」（42 ページ）を参照してください。

ドキュメントを開く/閉じる

ドキュメントを開く処理は、実に簡単なことに見えるかも知れません。アプリケーションは、`Documents`ディレクトリ以下にある、所定の拡張子のファイルを検索して表示し、どれを開くかユーザに選択させます。しかし、iCloudストレージがからむと少々複雑になります。ドキュメントの保存場所は、サンドボックス内の`Documents`ディレクトリ以下と、iCloudコンテナディレクトリの`Documents`ディレクトリ以下の、*2通り*があるからです。

アプリケーションのドキュメント検索

iCloudストレージに置かれたドキュメントリストを取得するためには、メタデータクエリを実行しなければなりません。これは`NSMetadataQuery`クラスのインスタンスです。作成した`NSMetadataQuery`オブジェクトに、検索範囲と述語を設定します。iCloudストレージの場合、検索範囲は`NSMetadataQueryUbiquitousDocumentsScope`となります。述語は`NSPredicate`オブジェクトで、この場合、ファイル拡張子により検索範囲をしぼり込む旨を表します。クエリを実行する前に、`NSMetadataQueryDidFinishGatheringNotification`および`NSMetadataQueryDidUpdateNotification`の通知を監視するよう登録します。この通知を受け取ったメソッドが、クエリ結果を処理します。

リスト4-3に、メタデータクエリを作成、実行して、iCloudモバイルコンテナにあるドキュメントリストを取得するコード例を示します。メソッドではまず、ユーザが設定したドキュメントの置き場所を確認します（この場合は`documentsInCloud`プロパティ）。これがモバイルコンテナであれば、メタデータクエリを実行します。一方、アプリケーションサンドボックスであれば、アプリケーションの`Documents`ディレクトリの中身を順次調べ、ローカルドキュメントファイルの名前と位置を取得します。

リスト 4-3 ドキュメントの保存場所を取得（ローカルに格納/iCloudストレージ）

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

```
    // ナビゲーション項目AddおよびEditをここでセットアップ...


   if (self.documentsInCloud) {
        _query = [[NSMetadataQuery alloc] init];
        [_query setSearchScopes:[NSArray
arrayWithObjects:NSMetadataQueryUbiquitousDocumentsScope, nil]];
        [_query setPredicate:[NSPredicate predicateWithFormat:@"%K LIKE '*.txt'",
 NSMetadataItemFSNameKey]];
        NSNotificationCenter* notificationCenter = [NSNotificationCenter
defaultCenter];
        [notificationCenter addObserver:self selector:@selector(fileListReceived)
            name:NSMetadataQueryDidFinishGatheringNotification object:nil];
        [notificationCenter addObserver:self selector:@selector(fileListReceived)
            name:NSMetadataQueryDidUpdateNotification object:nil];
        [_query startQuery];
    } else {
        NSArray* localDocuments = [[NSFileManager defaultManager]
contentsOfDirectoryAtPath:
            [self.documentsDir path] error:nil];
        for (NSString* document in localDocuments) {
            [_fileList addObject:[[[FileRepresentation alloc]
initWithFileName:[document lastPathComponent]
                url:[NSURL fileURLWithPath:[[self.documentsDir path]
                stringByAppendingPathComponent:document]]] autorelease]];
        }
    }
}
```

この例で、述語は「`@"%K LIKE '*.txt'"`」となっています。これは、拡張子が「`txt`」（このアプリケーションのドキュメントファイルの拡張子）である、すべてのファイル名（`NSMetadataItemFSNameKey`キー）を返す旨を表します。

1回目のクエリ実行後も、更新があれば、リスト 4-3に現れる、通知を受けるメソッド（`fileListReceived`）が再び起動されます。リスト4-4に、このメソッドの実装例を示します。ユーザが選択した後で更新があった旨の通知が届いた場合も、改めて選び直す必要がないよう保存されています。

リスト 4-4　iCloudストレージ上のドキュメントに関する情報の収集

```
-(void)fileListReceived {


 NSString* selectedFileName=nil;
    NSInteger newSelectionRow = [self.tableView indexPathForSelectedRow].row;
    if (newSelectionRow != NSNotFound) {
        selectedFileName = [[_fileList objectAtIndex:newSelectionRow] fileName];
    }
    [_fileList removeAllObjects];
    NSArray* queryResults = [_query results];
    for (NSMetadataItem* result in queryResults) {
        NSString* fileName = [result valueForAttribute:NSMetadataItemFSNameKey];
        if (selectedFileName && [selectedFileName isEqualToString:fileName]) {
            newSelectionRow = [_fileList count];
        }
       [_fileList addObject:[[[FileRepresentation alloc] initWithFileName:fileName
            url:[result valueForAttribute:NSMetadataItemURLKey]] autorelease]];
    }
    [self.tableView reloadData];
    if (newSelectionRow != NSNotFound) {
        NSIndexPath* selectionPath = [NSIndexPath indexPathForRow:newSelectionRow
 inSection:0];
        [self.tableView selectRowAtIndexPath:selectionPath animated:NO
scrollPosition:UITableViewScrollPositionNone];
    }
}
```

このアプリケーション例では、カスタムモデルオブジェクトの配列（`_fileList`）に、各アプリケーションドキュメントの名前とファイルURLをカプセル化しています(`FileRepresentation`はこのオブジェクトのカスタムクラス）。ルートビューコントローラは、プレーンテーブルビューにドキュメント名を表示します。

注意 メタデータクエリを実行するのは、アプリケーションがフォアグラウンドで動作している間だけにしてください。バックグラウンドになったら、クエリを停止しなければなりません。

ドキュメントファイルをiCloudからダウンロード

メタデータクエリによりiCloud上にあるドキュメントを調べると、クエリ結果は、ドキュメントファイルを表すプレースホルダ項目（`NSMetadataItem`オブジェクト）の形で得られます。ここには、URL、修正日など、ファイルに関するメタデータが収容されています。ドキュメントファイルは、iCloudコンテナディレクトリにはありません。

実際のドキュメントデータは、次のいずれかが起こった時点でダウンロードされます。

- アプリケーションが当該ファイルを開き、またはアクセスしようとしたとき（`openWithCompletionHandler:`の呼び出しなど）。
- アプリケーションが`NSFileManager`の`startDownloadingUbiquitousItemAtURL:error:`メソッドを呼び出して、明示的にデータをダウンロードしようとしたとき。

大容量のドキュメントファイルをiCloudからダウンロードしようとすると、ドキュメントデータの表示に相当の時間がかかるので、ダウンロード中である旨（「ロード中」、「更新中」など）と、まだアクセスできない旨を表示する必要があります。ダウンロードが終わったらこの表示を消してください。

ドキュメントを開く

例として示すドキュメントベースアプリケーションでは、既存のドキュメントを表に列挙しています。ユーザが開きたいドキュメントをタップすると、`UITableView`はそのデリゲートの`tableView:didSelectRowAtIndexPath:`メソッドを起動します。このメソッドの実装例（リスト4-5）は、典型的なナビゲーションパターンになっています。ルートビューコントローラは、次のビューコントローラを順次割り当て（この場合はドキュメントデータを表示するビューコントローラ）、実データ（この場合はドキュメントのファイルURL）を渡して初期化します。デバイスイディオムがiPadかiPhone（あるいはiPhone touch）か応じて、ルートビューコントローラはビューコントローラを、分割ビューに追加するか、またはナビゲーションコントローラのスタックにプッシュします。

リスト 4-5 ドキュメントを開く要求に対する応答

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

{
```

```
    [self selectFileAtIndexPath:indexPath create:NO];
}


-(void)selectFileAtIndexPath:(NSIndexPath*)indexPath create:(BOOL)create
{
    NSArray* fileList = indexPath.section == 0 ? _localFileList :
_ubiquitousFileList;
    DetailViewController* detailViewController = [[DetailViewController alloc]
        initWithFileURL:[[fileList objectAtIndex:indexPath.row] url]
createNewFile:create];


    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPad)
 {
        self.splitViewController.viewControllers =
            [NSArray arrayWithObjects:self.navigationController,
detailViewController, nil];
    }
    else {
        [self.navigationController pushViewController:detailViewController
animated:YES];
    }
    [detailViewController release];
}
```

初期化メソッド（コード例は省略）では、ドキュメントのビューコントローラ（この例では`DetailViewController`）が`UIDocument`のサブクラスのインスタンスを作成し、ファイルURLを渡して`initWithFileURL:`メソッドを呼び出すことにより初期化しています。また、作成したドキュメントオブジェクトを`document`プロパティに設定しています。

ドキュメントを開く処理の最後に、`UIDocument`オブジェクトの`openWithCompletionHandler:`メソッドを呼び出します。この例では、ドキュメントのビューコントローラは、`viewWillAppear:`の中でこのメソッドを呼び出しています（リスト4-6を参照）。このコードでは、すでに開いたものでないか、ドキュメントの状態を調べています。すでに開いているドキュメントを、改めて開く必要はありません。

リスト 4-6　ドキュメントを開く

```
-(void)viewWillAppear:(BOOL)animated {
```

```
    [super viewWillAppear:animated];
    if (_createFile) {
        [self.document saveToURL:self.document.fileURL
forSaveOperation:UIDocumentSaveForCreating
            completionHandler:^(BOOL success) {
                _textView.text = self.document.text;
        }];
        _createFile = NO;
    }
    else {
        if (self.document.documentState & UIDocumentStateClosed) {
            [self.document openWithCompletionHandler:nil];
        }
    }
}
```

`openWithCompletionHandler:`が呼び出されると、`UIDocument`はデータをドキュメントファイルから読み込み、ドキュメントオブジェクト自身はそのモデルオブジェクトをデータから作成します。この一連の処理の最後に、`openWithCompletionHandler:`メソッドの完了ハンドラが実行されます。この例のビューコントローラには完了ブロックが実装されていませんが、完了ハンドラは、ドキュメントデータをドキュメントのビューに代入し、表示するために使うことがあります（`DetailViewController`がドキュメントビューを更新する代わりに実行している処理については、リスト 3-4 （27 ページ）および関係する本文を参照）。

ドキュメントを閉じる

ドキュメントを閉じるためには、ドキュメントオブジェクトの`closeWithCompletionHandler:`メソッドを実行します。このメソッドは、必要ならばドキュメントデータを保存した後、完了ハンドラ（このメソッドに渡す唯一のパラメータ）を実行します。

ドキュメントを閉じる時点としては、ドキュメントのビューコントローラが消えたとき（ユーザが「戻る」ボタンをタップしたなど）が適切です。ビューコントローラのビューが消える直前に、`viewWillDisappear:`メソッドが起動されます。ビューコントローラのサブクラスでは、このメソッドをオーバーライドして、ドキュメントオブジェクトの`closeWithCompletionHandler:`を呼び出すこともできます（リスト 4-7を参照）。

リスト 4-7　ドキュメントを閉じる

```
-(void)viewDidDisappear:(BOOL)animated {
    [super viewDidDisappear:animated];
    [self.document closeWithCompletionHandler:nil];
}
```

ドキュメントをiCloudストレージに、またはiCloudストレージから移動

「ドキュメントファイルの保存場所の設定」（32 ページ）で述べたように、アプリケーションには、ドキュメントをすべてローカルファイルシステム（アプリケーションのサンドボックス）に保存するか、iCloud（コンテナディレクトリ）に保存するか、の選択肢を用意するべきです。これを環境設定として保存しておき、ドキュメントを保存したり開いたりする際に参照します。ユーザがこの環境設定を変更した場合、その変更内容に応じ、アプリケーションサンドボックスのドキュメントファイルをすべてiCloudに、またはその逆に移動する必要があります。

iCloudコンテナディレクトリの場所の取得

ドキュメントファイルをローカルストレージからiCloudコンテナディレクトリの`Documents`サブディレクトリに移動する際、ファイル名は変えません。変更するのはファイルURLの先頭（`Documents`よりも前の）部分だけです。パスのこの部分を取得するためには、`NSFileManager`の`URLForUbiquityContainerIdentifier:`メソッドを、有効なiCloudコンテナ識別子を渡して呼び出す必要があります。アプリケーションのコンテナ識別子（チームIDとアプリケーションバンドルIDを、ピリオドを挟んで連結したもの）は、Xcode上で、対象の「Summary」ビューの「Identifier」フィールドからコピーできます（これはアプリケーションの主コンテナ識別子です。追加のコンテナディレクトリを設けることも可能です）。次のように、コンテナ識別子を表す文字列定数を宣言しておくのもよいでしょう。

```
static NSString *UbiquityContainerIdentifier =
@"A93A5CM278.com.acme.document.billabong";
```

リスト 4-8に示す2つのメソッドは、iCloudコンテナ識別子を取得し、「/Documents」という文字列を付加しています。

リスト 4-8　iCloudコンテナディレクトリURLの取得

```
-(NSURL*)ubiquitousContainerURL {
```

```
    return [[NSFileManager defaultManager]
URLForUbiquityContainerIdentifier:UbiquityContainerIdentifier];
}

-(NSURL*)ubiquitousDocumentsDirectoryURL {
    return [[self ubiquitousContainerURL] URLByAppendingPathComponent:@"Documents"];
}
```

注意 アプリケーションサンドボックスのベースURLを取得する例は、「ファイルURLの組み立て、ドキュメントファイルの保存」（34 ページ）を参照してください。

ドキュメントをiCloudストレージに移動

プログラム上、ドキュメントをiCloudストレージに置く処理は、`NSFileManager`の`setUbiquitous:itemAtURL:destinationURL:error:`を呼び出すだけでできてしまいます。このメソッドに、アプリケーションサンドボックス内と、アプリケーションのiCloudコンテナディレクトリにおける、ドキュメントファイルのファイルURL（ソースURLとデスティネーションURL）を渡します。第1パラメータはブール値で、`YES`でなければなりません。

Important `setUbiquitous:itemAtURL:destinationURL:error:`メソッドをアプリケーションのメインスレッドから呼び出してはなりません（特にドキュメントを閉じていない場合）。このメソッドは、指定されたファイルに対して協調書き込みを行うので、メインスレッドからこのメソッドを呼び出すと、ファイルを監視しているファイルプレゼンタとの間でデッドロックが起こる恐れがあります（さらに、完了まで長い時間がかかり、アプリケーションが応答しなくなる恐れもあります）。代わりに、メインスレッドキュー以外のディスパッチキューで動作するブロック内で、このメソッドを呼び出すようにしてください。処理完了後、メインスレッドにメッセージを送ることにより、アプリケーションが管理するデータ構造を更新して、整合性を保つことができます。

リスト4-9に、アプリケーションサンドボックス内からiCloudストレージにドキュメントを移動する例を示します。このアプリケーション例では、ユーザがストレージ場所（iCloudまたはローカル）を変更すると、アプリケーションサンドボックス内の各ドキュメントファイルについてこのメソッドが呼び出されます。メソッドは大まかに、次の3つの部分から成ります。

- ソースURL、デスティネーションURLを組み立てる。
- （副ディスパッチキューにおいて）`setUbiquitous:itemAtURL:destinationURL:error:`メソッドを呼び出し、結果を表すブール値（`success`）をキャッシュする。この値は、ドキュメントファイルをiCloudコンテナディレクトリに、正常に移動できたかどうかを表す。

- （メインディスパッチキューにおいて）呼び出しに成功したら、ドキュメントのモデルオブジェクトおよびその表示を更新する。失敗であれば、エラー情報をログに出力する（など適切な処置を施す）。

リスト 4-9　ドキュメントファイルをローカルストレージからiCloudストレージに移動

```
- (void)moveFileToiCloud:(FileRepresentation *)fileToMove {
    NSURL *sourceURL = fileToMove.url;
    NSString *destinationFileName = fileToMove.fileName;
    NSURL *destinationURL = [self.documentsDir
URLByAppendingPathComponent:destinationFileName];

    dispatch_queue_t q_default;
    q_default = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(q_default, ^(void) {
        NSFileManager *fileManager = [[[NSFileManager alloc] init] autorelease];
        NSError *error = nil;
        BOOL success = [fileManager setUbiquitous:YES itemAtURL:sourceURL
            destinationURL:destinationURL error:&error];
        dispatch_queue_t q_main = dispatch_get_main_queue();
        dispatch_async(q_main, ^(void) {
            if (success) {
              FileRepresentation *fileRepresentation = [[FileRepresentation alloc]
                    initWithFileName:fileToMove.fileName url:destinationURL];
                [_fileList removeObject:fileToMove];
                [_fileList addObject:fileRepresentation];
                NSLog(@"moved file to cloud: %@", fileRepresentation);
            }
            if (!success) {
                 NSLog(@"Couldn't move file to iCloud: %@", fileToMove);
            }
        });
    });
}
```

ドキュメントをiCloudストレージから削除

ドキュメントファイルをiCloudコンテナディレクトリからアプリケーションサンドボックスのDocumentsディレクトリに移動する場合も、「ドキュメントをiCloudストレージに移動」（43 ページ）と同様の手順ですが、ソースURL（iCloudコンテナディレクトリのドキュメントファイル）とデスティネーションURL（アプリケーションサンドボックスのドキュメントファイル）の関係が逆になります。さらに、setUbiquitous:itemAtURL:destinationURL:error:メソッドの第1パラメータはNOとなります。リスト 4-10にこの手順の実装例を示します。iCloudコンテナディレクトリ内の各ファイルについて呼び出され、アプリケーションサンドボックスに移動する処理を行います。

リスト 4-10　ドキュメントファイルをiCloudストレージからローカルストレージに移動

```
- (void)moveFileToLocal:(FileRepresentation *)fileToMove {
    NSURL *sourceURL = fileToMove.url;
    NSString *destinationFileName = fileToMove.fileName;
    NSURL *destinationURL = [self.documentsDir
URLByAppendingPathComponent:destinationFileName];

    dispatch_queue_t q_default;
    q_default = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
    dispatch_async(q_default, ^(void) {
        NSFileManager *fileManager = [[[NSFileManager alloc] init] autorelease];
        NSError *error = nil;
        BOOL success = [fileManager setUbiquitous:NO itemAtURL:sourceURL
destinationURL:destinationURL
            error:&error];
        dispatch_queue_t q_main = dispatch_get_main_queue();
        dispatch_async(q_main, ^(void) {
            if (success) {
              FileRepresentation *fileRepresentation = [[FileRepresentation alloc]
                    initWithFileName:fileToMove.fileName url:destinationURL];
                [_fileList removeObject:fileToMove];
                [_fileList addObject:fileRepresentation];
                NSLog(@"moved file to local storage: %@", fileRepresentation);
            }
            if (!success) {
                NSLog(@"Couldn't move file to local storage: %@", fileToMove);
            }
```

```
        });
    });
}
```

ドキュメント状態の変化の監視とエラー処理

ドキュメントの状態は実行時に変化することがあります。状態を調べることにより、エラーが生じた、版の食い違いが生じたなど、異常を知ることができます。UIDocumentに宣言されている（UIDocumentState型の）定数を使って、ドキュメントの状態を表すほか、状態が変化した場合には適切な値をdocumentStateプロパティに設定することができます。 表 4-1に、状態を表す定数を示します。

表 4-1　UIDocumentStateの定数

ドキュメントの状態を表す定数	意味
UIDocumentStateNormal	ドキュメントは開いており、版の食い違いその他の問題は発生していない。
UIDocumentStateClosed	ドキュメントは閉じている。UIDocumentがドキュメントを開けない場合にこの状態になる。この場合、ドキュメントプロパティが正しくないかも知れない。
UIDocumentStateInConflict	食い違いが生じている版がある。
UIDocumentStateSavingError	エラーのためUIDocumentがドキュメントを保存できない。
UIDocumentState-EditingDisabled	ユーザが安全にドキュメントを編集できる状態ではない。

UIDocumentは、ドキュメントの状態が変化したとき、UIDocumentStateChangedNotification型の通知も発信します。アプリケーションはこの通知を監視し、適切に処理しなければなりません。監視の登録をする場所としては、ドキュメントのビューコントローラの初期化メソッドが適しています（リスト 4-11を参照）。この場合、監視するのはビューコントローラです。

リスト 4-11　UIDocumentStateChangedNotification通知を監視する旨の登録を追加

```
-(id)initWithFileURL:(NSURL*)url createNewFile:(BOOL)createNewFile {
    NSString* nibName = [[UIDevice currentDevice] userInterfaceIdiom] ==
```

```
        UIUserInterfaceIdiomPad ? @"DetailViewController_iPad" :
@"DetailViewController_iPhone";
    self = [super initWithNibName:nibName bundle:nil];
    if (self) {
        _document = [[ImageNotesDocument alloc] initWithFileURL:url];
        // その他のコード...
        [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(documentStateChanged)
            name:UIDocumentStateChangedNotification object:_document];
    }
    return self;
}
```

クラスの`dealloc`メソッドの通知センタからは、監視の登録を削除してください。

ドキュメントの状態が変化すると、`UIDocument`は`UIDocumentStateChangedNotification`通知を発信し、通知センタはこれを、通知メソッド（この例では`documentStateChanged`）を起動することにより配布します。リスト4-12では、ビューコントローラは通知が届くと、現在の状態を`documentState`プロパティから取得し、評価しています。状態が`UIDocumentStateEditingDisabled`であれば、キーボードを非表示にして使えないようにします。版の食い違いがある（`UIDocumentStateInConflict`）場合は、「Show Conflicts」ボタンをドキュメントビューのツールバーに表示します（このときの対処法については「ドキュメントの版の食い違い解消」（54 ページ）を参照）。

リスト 4-12　現在のドキュメント状態の評価

```
-(void)documentStateChanged {
    UIDocumentState state = _document.documentState;
    [_statusView setDocumentState:state];
    if (state & UIDocumentStateEditingDisabled) {
        [_textView resignFirstResponder];
    }
    if (state & UIDocumentStateInConflict) {
        [self showConflictButton];
    }
    else {
        [self hideConflictButton];
        [self dismissModalViewControllerAnimated:YES];
```

```
    }
}
```

通知の処理メソッドは、プライベートビュークラスに実装されたsetDocumentState:メソッドも呼び出します。このメソッド（リスト4-13）は、ドキュメントの状態に応じて、ドキュメントビューのツールバー上にある表示を切り替えます。

リスト 4-13　状態に応じてドキュメントのユーザインターフェイスを更新

```
-(void)setDocumentState:(UIDocumentState)documentState {
    if (documentState & UIDocumentStateSavingError) {
        self.unsavedLabel.hidden = NO;
        self.circleView.image = [UIImage imageNamed:@"Red"];
    }
    else {
        self.unsavedLabel.hidden = YES;
        if (documentState & UIDocumentStateInConflict) {
            self.circleView.image = [UIImage imageNamed:@"Yellow"];
        }
        else {
            self.circleView.image = [UIImage imageNamed:@"Green"];
        }
    }
}
```

ドキュメントを保存できない（UIDocumentStateSavingError）場合、ビューコントローラは状態インジケータを赤色にし、隣に「Unsaved」と表示します。版の食い違いがあれば、状態インジケータを黄色にします（先に説明した「Show Conflicts」ボタンの表示に加えて）。そうでなければ状態インジケータは緑色です。

ドキュメントの削除

ユーザがドキュメントを作成できるようにしたのと同様に、選択したドキュメントを削除する機能も必要になるでしょう。ドキュメントを削除するためには、次の3つの処理が必要です。

- ドキュメントファイルをストレージ（ローカルサンドボックス、またはiCloudコンテナディレクトリ）から削除する。
- ドキュメントデータを表すために使っていたモデルオブジェクトをメモリ上から消去する。
- ドキュメントビューに表示していたドキュメントデータを削除する。

ドキュメントをストレージから削除する際には、読み書き処理の際に`UIDocument`がしていたのと同様の注意が必要です。すなわち、削除はバックグラウンドキュー上で非同期に行い、File Coordination APIを使わなければなりません。リスト 4-14にそのコード例を示します。バックグラウンドキュー上にタスクをディスパッチします。このタスクは、`NSFileCoordinator`オブジェクトを作成し、その`coordinateWritingItemAtURL:options:error:byAccessor:`メソッドを呼び出します。このメソッドの`byAccessor`ブロックから、`NSFileManager`のファイル削除メソッドである`removeItemAtURL:error:`を呼び出しています。

リスト 4-14　選択されたドキュメントの削除

```
-(void)tableView:(UITableView *)tableView
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
       forRowAtIndexPath:(NSIndexPath *)indexPath {
    NSMutableArray* fileList = nil;
    if (indexPath.section == 0) {
        fileList = self.localFileList;
    }
    else {
        fileList = self.ubiquitousFileList;
    }
    NSURL* fileURL = [[fileList objectAtIndex:indexPath.row] url];
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
 ^(void) {
        NSFileCoordinator* fileCoordinator = [[NSFileCoordinator alloc]
initWithFilePresenter:nil];
        [fileCoordinator coordinateWritingItemAtURL:fileURL
options:NSFileCoordinatorWritingForDeleting
            error:nil byAccessor:^(NSURL* writingURL) {
            NSFileManager* fileManager = [[NSFileManager alloc] init];
            [fileManager removeItemAtURL:writingURL error:nil];
        }];
    });
    [fileList removeObjectAtIndex:indexPath.row];
```

```
    [tableView deleteRowsAtIndexPaths:[[NSArray alloc] initWithObjects:&indexPath
 count:1]
        withRowAnimation:UITableViewRowAnimationLeft];
}
```

この例では、テーブルビューが編集モードであるときにユーザが「Delete」ボタンをタップすると、このメソッドが起動されるようになっています。

状態変化の追跡、取り消し処理

UIDocumentクラスは「保存操作なし」モデルに従っているため、ドキュメントデータは頻繁に自動保存され、ユーザが意識的にドキュメントを保存する必要はありません。UIDocumentにはこのモデルに基づくさまざまな処理が実装されていますが、実際にこの機能が働くためには、ドキュメントベースアプリケーション側もある役割を果たす必要があります。

UIKitがドキュメントデータを自動保存する仕組み

UIKitフレームワークに実装されている「保存操作なし」モデルは、保存が必要なドキュメントに印をつける仕組みと、フレームワークがその状態を調べる頻度を調整する仕組みの、2つの部分に分かれます。UIKitは定期的に、UIDocumentのhasUnsavedChangesメソッドを呼び出し、戻り値を評価します。値がYESであれば、ドキュメントデータをドキュメントファイルに保存します。hasUnsavedChangesの値を調べる頻度は、ユーザの入力速度など、いくつかの要因によって変わります。

ドキュメントベースアプリケーションは、取り消し/やり直し機能を実装する、あるいはドキュメントの状態変化を追跡することにより、hasUnsavedChangesで返される値を間接的に設定します。状態変化を追跡するためには、UIDocumentChangeDone（UIDocumentChangeKind型の定数）を渡してupdateChangeCount:メソッドを呼び出す必要があります。アプリケーションが取り消しアクションを登録し、undoまたはredoメッセージをドキュメントの取り消しマネージャに送信すると、UIDocumentは代わりにupdateChangeCount:を呼び出します。

取り消し/やり直しはアプリケーションの差別化にもつながる機能なので、実装することをお勧めします。

取り消し/やり直しの実装

取り消し/やり直しの機能をアプリケーションに実装する際は、『*Undo Architecture*』に示した手順や推奨事項に従ってください。なお、UIDocumentにはundoManagerプロパティが定義されています。デフォルトのNSUndoManagerオブジェクトは、このプロパティにアクセスすることにより取得できます。また、独自のNSUndoManagerオブジェクトをこのプロパティに設定することも可能です。取り消しマネージャは、UIDocumentオブジェクトにプロパティの形で関連づけて、状態変化を追跡し、ドキュメントデータを自動保存できるようにする必要があります。

リスト 5-1に、あるテキストフィールドを対象とした取り消し/やり直し機能の実装例を示します。

リスト 5-1　テキストフィールドを対象とした取り消し/やり直し機能の実装

```
- (void)textFieldDidEndEditing:(UITextField *)textField {
    self.undoButton.enabled = YES;
    self.redoButton.enabled = YES;

    if (textField.tag == 1) {
        [self setLocationText:textField.text];
    }
    // ほかのテキストフィールドを対象とするコード...
}

- (void)setLocationText:(NSString *)newText {
    NSString *currentText = _document.location;
    if (newText != currentText) {
        [_document.undoManager registerUndoWithTarget:self
            selector:@selector(setLocationText:)
            object:currentText];
        _document.location = newText;
        self.locationField.text = newText;
    }
}

- (IBAction)handleUndo:(id)sender {
    [_document.undoManager undo];
    if (![_document.undoManager canUndo]) self.undoButton.enabled = NO;
}

- (IBAction)handleRedo:(id)sender {
    [_document.undoManager redo];
    if (![_document.undoManager canRedo]) self.redoButton.enabled = NO;
}
```

状態変化の追跡処理の実装

取り消し/やり直しの代わりに状態変化の追跡処理を実装するためには、コード中の適当な箇所で、`UIDocument`の`updateChangeCount:`メソッドを呼び出します。取り消しアクションの登録と同様、ユーザの入力に応じてドキュメントのモデルオブジェクトを更新する時点で呼び出すようにするとよいでしょう。このときに渡すパラメータは`UIDocumentChangeDone`定数です。

リスト 5-2に、テキストビューの状態に変化があると呼び出される`UITextViewDelegate`メソッドから、`updateChangeCount:`を呼び出すコード例を示します。

リスト 5-2　ドキュメントの変更回数を更新

```
-(void)textViewDidChange:(UITextView *)textView {
    _document.documentText = textView.text;
    [_document updateChangeCount:UIDocumentChangeDone];
}
```

ドキュメントの版の食い違い解消

iCloudの世界では、ドキュメントベースアプリケーションを複数のデバイスやデスクトップシステムにインストールしていると、同じドキュメントでも、内容が食い違う異なる版ができる恐れがあります。先に説明したように、アプリケーションはドキュメントファイルをローカルコンテナディレクトリ上で更新し、（通常は即座に）iCloudに転送します。しかし、即座に転送しなければどうなるでしょうか。たとえば、あるドキュメントをMac OS X用のアプリケーションで編集中、同じドキュメントをiPadでも編集し、その際、デバイスは機内（Airplane）モードになっていたとします。機内モードを解除すると、ドキュメントに施したローカルな変更はiCloudに転送されます。このときiCloudは、食い違いがあることを認識してアプリケーションに通知します。

版の食い違いの認識

「ドキュメント状態の変化の監視とエラー処理」（46 ページ）で説明したように、アプリケーションは版の食い違いを認識するため、`UIDocumentStateChangedNotification`通知を監視しています。`documentState`プロパティが`UIDocumentStateInConflict`に変われば、同じドキュメントに複数の版が存在することになります。アプリケーションは、ユーザの介入を求め、または求めずに、できるだけ早期にこの食い違いを解消しなければなりません。

版の食い違いは、`NSFileVersion`クラスの、2つのクラスメソッドで認識します。`currentVersionOfItemAtURL:`メソッドは、*現行ファイル*として参照されるものに対応する`NSFileVersion`オブジェクトを返します。現行ファイルは、現在の「conflict winner」などを基準としてiCloudが選択するため、どのデバイスから見ても同じです。`unresolvedConflictVersionsOfItemAtURL:`メソッドを呼び出すことにより、`NSFileVersion`オブジェクトの配列が得られます。この各オブジェクトを*食い違っている版*と呼び、それぞれ、指定されたURLにあるファイルに関する、未解決の食い違いを表します。`NSFileVersion`オブジェクトには、食い違いの解消に役立つ情報（修正日、ローカライズされたドキュメント名、保存に用いたコンピュータのローカライズされた名前など）があります。

版の食い違いを解消する戦略

アプリケーションは、3つの戦略のいずれかにより、版の食い違いを解消できます。

- 食い違っている部分をマージする。

- 何らかの基準（最終修正日など）により、いずれかの版を選択する。
- 食い違いの状況を表示し、ユーザに選択してもらう。

どの戦略が適しているかは、ドキュメントデータによって異なります。整合性を損なうことなく内容をマージできるのであれば、この戦略を採用してください。あるいは、データを失う恐れがないのであれば、修正日が最も新しい版を選択するとよいでしょう。

通常はユーザの介入なしに解決するよう試みるべきですが、それが不可能なアプリケーションもあります。ユーザに判断を委ねる方針であれば、食い違いの詳しい状況を表示し、選択ボタンその他、適当なインターフェイスを用意してください。「例: ユーザに版を選択させる戦略」（56 ページ）に、ユーザに選択を委ねるコード例を示します。

食い違いを解消した旨をiOSに通知する方法

アプリケーションまたはユーザが、いずれかの版を選択して食い違いを解消した場合、アプリケーションは次の処理を実行する必要があります。

- 食い違っている版を選択した場合は、現行のドキュメントファイルをその版で置き換える。

  版を表す`NSFileVersion`の`replaceItemAtURL:options:error:`メソッドを、現行のファイルURLを渡して呼び出します。
- 食い違っている版を選択した場合は、置き換わった内容が表示されるよう、ドキュメントの「復元」操作を行う。

  `UIDocument`の`revertToContentsOfURL:completionHandler:`メソッドを、現行のファイルURLを渡して呼び出します。
- 食い違いを起こしていた版すべてについて、ドキュメントのファイルURLとの関連を切る。

  `NSFileVersion`の`removeOtherVersionsOfItemAtURL:error:`メソッドを、ドキュメントのファイルURLを渡して呼び出します。
- 食い違いを起こしていた版それぞれに解決済みである旨の印をつけて、iOSが再び食い違いと認識しないようにする。

  食い違いを起こしていた各版の`NSFileVersion`の`resolved`プロパティを`YES`とします。この処理は最後に実行してください。

例：ユーザに版を選択させる戦略

ドキュメントベースアプリケーションの例として、簡単なテキストエディタを取り上げましょう。このようなアプリケーションの場合、食い違っている版との差分を調べてマージする処理は非常に難しく、できたとしても、ユーザにとって望ましいとは限りません。修正日が最新の版を選択する、という戦略も考えられますが、やはりこれがユーザの望んでいる方法とは限りません。この場合最もよいのは、ドキュメントの内容を熟知しているユーザ自身に選択させる、という戦略でしょう。

リスト 6-1に、「ドキュメント状態の変化の監視とエラー処理」（46 ページ）に示したコードを再掲します。これはドキュメントのビューコントローラのメソッドで、ドキュメントの状態が変化した場合に`UIDocument`が送信する、`UIDocumentStateChangedNotification`通知を処理するものです。状態が`UIDocumentStateInConflict`になると、ビューコントローラはカスタム状態ビューに「Resolve Conflicts」ボタンを表示します（また、状態インジケータを赤にします）。

リスト 6-1　版の食い違いを検知

```
-(void)documentStateChanged {
    UIDocumentState state = _document.documentState;
    [_statusView setDocumentState:state];
    if (state & UIDocumentStateEditingDisabled) {
        [_textView resignFirstResponder];
    }

    if (state & UIDocumentStateInConflict) {
        [self showConflictButton];            // &lt;------ 「Resolve Conflicts」ボタンを表示
    }
    else {
        [self hideConflictButton];
        [self dismissModalViewControllerAnimated:YES];
    }
}
```

ユーザがボタンをタップすると、UIKitはリスト 6-2に示すメソッドを起動します。このメソッドは、独自の食い違い解消ビューコントローラのビューを、モーダルダイアログとして表示します。

リスト 6-2　ドキュメントの食い違いを解消するユーザインターフェイスの表示

```
-(void)conflictButtonPushed
```

```
{
    ConflictResolverViewController* conflictResolver =
[[ConflictResolverViewController alloc]
        initWithURL:_document.fileURL delegate:self];
    [self presentViewController:conflictResolver animated:YES completion:nil];
    [conflictResolver release];
}
```

ConflictResolverViewControllerオブジェクトはページビューコントローラ（UIPageViewControllerオブジェクト）を作成して、現行ファイルのドキュメントと食い違っている版を、ユーザが比較、検証できるようにします。各ドキュメントビューのツールバーには「Select Version」ボタンが表示されます。ユーザがこのボタンをタップすると、選んだのが現行ファイルドキュメントか食い違っている版かに応じて、リスト 6-3に示したカスタムデリゲーションメソッドのいずれかが呼び出されます。

リスト 6-3　食い違いの解消

```
-(void)conflictResolver:(ConflictResolverViewController *)conflictResolver
       didResolveWithFileVersion:(NSFileVersion *)fileVersion {
    [self dismissViewControllerAnimated:YES completion:nil];
    [fileVersion replaceItemAtURL:_document.fileURL options:0 error:nil];
    [NSFileVersion removeOtherVersionsOfItemAtURL:_document.fileURL error:nil];
    [_document revertToContentsOfURL:_document.fileURL completionHandler:nil];
    NSArray* conflictVersions = [NSFileVersion
unresolvedConflictVersionsOfItemAtURL:_document.fileURL];
    for (NSFileVersion* fileVersion in conflictVersions) {
        fileVersion.resolved = YES;
    }
}


-(void)conflictResolverDidResolveWithCurrentVersion:(ConflictResolverViewController*)conflictResolver
 {
    [self dismissViewControllerAnimated:YES completion:nil];
    [NSFileVersion removeOtherVersionsOfItemAtURL:_document.fileURL error:nil];
    NSArray* conflictVersions = [NSFileVersion
unresolvedConflictVersionsOfItemAtURL:_document.fileURL];
    for (NSFileVersion* fileVersion in conflictVersions) {
```

```
            fileVersion.resolved = YES;
        }
    }
```

このメソッドは、「食い違いを解消した旨をiOSに通知する方法」（55ページ）に示した手順の例にもなっています。食い違っている版が選択された場合、デリゲートは、渡されたNSFileVersionオブジェクトのreplaceItemAtURL:options:error:メソッドを呼び出して、iCloudコンテナディレクトリにあるドキュメントファイルを、該当するドキュメントで置き換えます。次に、食い違っている各版に対応するNSFileVersionオブジェクトをすべて列挙した配列を取り上げ、各オブジェクトのresolvedプロパティをYESとします。続いて、NSFileVersionに対して、ドキュメントのファイルURLに関連づけられた、食い違っているほかの版をすべて削除するよう指示します。さらに、revertToContentsOfURL:completionHandler:を呼び出して、置き換わった内容が表示されるよう「復元」します。

2つ目のデリゲーションメソッドは、現行のドキュメントファイルが選択された場合の処理で、上記よりも単純です。食い違っている版のNSFileVersionオブジェクトすべてについて、resolvedプロパティをYESとした後、ドキュメントファイルURLに関連づけられた、食い違っている版をすべて削除します。

書類の改訂履歴

この表は「*iOS ドキュメントベース アプリケーション プログラミングガイド*」の改訂履歴です。

日付	メモ
2012-01-09	メタデータクエリ検索の述語に用いる演算子をLIKEに変更しました。
2011-10-12	iCloudストレージにドキュメントが統合されたiOSアプリケーションの作成方法について説明した新規ドキュメント。

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

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/