

ファイルシステムプロ グラミングガイド

目次

ファイルとディレクトリの概要 8

At a Glance 8

- ファイルシステムの編成は特定化される 8
- ファイルへの安全なアクセス 9
- ファイルへのアクセス方法はファイルタイプに依存 9
- システムインターフェイスがファイルの検出と管理をサポート 9
- ユーザは標準のシステムパネルを使用してファイルと対話 9
- ファイルの非同期の読み書き 10
- Finderと同様のファイルの移動、コピー、削除、管理 10
- ファイル関連の操作の最適化 10

関連項目 11

ファイルシステムの基礎 12

iOSのファイルシステムについて 12

- 島としてのアプリケーション 12
- iOS標準ディレクトリ：ファイルの保存場所 13
- アプリケーションファイルの置き場所 15

OS Xファイルシステムの概要 16

- ファイルの配置はドメインに依存 16
- OS X標準ディレクトリ：ファイルの保存場所 19
- サンドボックス内のOS Xアプリケーションファイルのコンテナ 21
- 非表示ファイルとディレクトリ：ユーザ体験の簡素化 21
- ファイルとディレクトリに代替名を割り当てられる 22
- Libraryディレクトリはアプリケーション固有のファイルを保存 23

iCloudファイルのストレージコンテナ 25

ファイルの内容のタイプの識別方法 26

セキュリティ：作成するファイルの保護 27

- サンドボックスは被害の拡大を制限 27
- すべてのファイルアクセスに適用される許可とアクセス制御リスト 28
- ディスクでのファイルの暗号化 28

同期によりファイル関連のコードの強度を保証 29

ファイル、並列処理、スレッドの安全性 29

ファイルとディレクトリへのアクセス 31

ファイルアクセスの正しい方法の選択	31
ファイルまたはディレクトリのパスの指定	33
ファイルシステムの項目の検索	35
ユーザへの項目の検索の要求	36
アプリケーションバンドルの項目の検索	36
標準ディレクトリの項目の検索	37
ブックマークを使用したファイルの検索	38
ディレクトリの内容の列挙	40
ディレクトリのファイルの逐次列挙	40
1回のバッチ操作によるディレクトリの内容の取得	43
アプリケーション固有のファイルの保存場所の決定	44
ファイルとディレクトリのアクセスに関するヒント	45
ファイル操作のパフォーマンスの遅延	45
セキュアなコーディング方式の使用	45
パスの大文字小文字区別を想定	46
新しいファイルのファイル名拡張子をインクルード	47
項目を表示する場合の表示名の使用	47
バックグラウンドスレッドからの安全なファイルアクセス	47
ファイルコーディネータとファイルプレゼンタの役割	48
NSDocumentとUIDocumentを使用したファイルコーディネータとファイルプレゼンタの処理	48
ファイルコーディネータを使用した安全な読み書き操作の保証	49
ファイルプレゼンタにより監視するファイルの選択	49
ファイルプレゼンタオブジェクトの実装	50
ファイルプレゼンタのシステムへの登録	51
ファイルコーディネータによるファイル変更の開始	51
iCloudのファイル管理	53
iCloudのドキュメントの保存と使い方	53
iCloudストレージAPI	54
iCloudのドキュメントの操作	55
ドキュメントをiCloudストレージに移動	57
iCloudのドキュメントの検索	58
ファイルの版の食い違いの処理	59
iCloudストレージの信頼性のある使い方	61
OpenおよびSaveパネルの使用	62
Openパネル：既存のファイルまたはディレクトリの取得	62
表示予定のユーザドキュメントを新しいウインドウで開く	62
すでに開いているウインドウのファイルとディレクトリの選択	64

Saveパネル：新しいファイル名の取得	65
ユーザが選択できるファイルタイプのフィルタによる制限	67
OpenおよびSaveパネルへのアクセサリビューの追加	68
アクセサリビューの内容の定義	68
実行時のアクセサリビューのロードと設定	70
ファイルとディレクトリの管理	72
新しいファイルとディレクトリのプログラムによる作成	72
ディレクトリの作成	72
新しいファイルの作成	74
ファイルとディレクトリのコピーと移動	75
ファイルとディレクトリの削除	76
非表示ファイルの作成と処理	77
ファイルコーデネータを使用せずにファイルを読み書きする方法	78
ファイルの非同期の読み書き	78
ストリームを使用したファイル全体の線形処理	78
GCDを使用したファイルの処理	79
ファイルの同期的な読み書き	85
メモリの内容の構築とディスクへの書き込み操作の同時実施	85
NSFileHandleを使用したファイルの読み書き	86
ディスクの読み書きのPOSIXレベルの管理	87
ファイルメタデータ情報の取得と設定	88
ファイルラッパーのファイルコンテナとしての使用	90
ファイルラッパーの操作	90
ディレクトリラッパーの操作	91
パフォーマンスのヒント	93
コード内の検出項目	93
最新のファイルシステムインターフェイスの使用	94
一般的なヒント	94
システム独自のファイルキャッシュメカニズム	95
Zero-Fill遅延によるセキュリティの強化と高コスト化	96
OS X Libraryディレクトリの詳細	98
ファイルシステムの詳細	102
サポートされるファイルシステム	102
Finder	103

ファイル名の並べ替え規則	103
ファイルとディレクトリの表示規則	104
ファイルタイプとクリエータコード	105
OS Xファイルシステムのセキュリティ	105
セキュリティスキーム	105
特別なユーザとグループ	117
ネットワークファイルシステム	119
iOSのファイルシステムのセキュリティ	121
書類の改訂履歴	123

図、表、リスト

ファイルシステムの基礎 12

- 図 1-1 各iOSアプリケーションは専用のサンドボックス内で動作 13
- 図 1-2 ローカルOS Xファイルシステム 18
- 表 1-1 iOSアプリケーションの共通に使用されるディレクトリ 14
- 表 1-2 OS Xの共通に使用されるディレクトリ 19
- 表 1-3 Libraryディレクトリの主要なサブディレクトリ 24
- 表 1-4 主要なクラスと技術のスレッドの安全性 30

ファイルとディレクトリへのアクセス 31

- 表 2-1 特別なルーチンを用いるファイルタイプ 31
- リスト 2-1 アプリケーションサポートディレクトリの項目のURLの作成 37
- リスト 2-2 URLの永続形式への変換 38
- リスト 2-3 永続ブックマークのURL形式への変換 39
- リスト 2-4 ディレクトリの内容の列挙 41
- リスト 2-5 最近修正されたファイルの検索 42
- リスト 2-6 ディレクトリの項目のリストを一度で取得する 43

iCloudのファイル管理 53

- 図 4-1 ドキュメントに施した変更をiCloudに反映 54

OpenおよびSaveパネルの使用 62

- リスト 5-1 Openパネルのユーザへの表示 63
- リスト 5-2 Openパネルのウインドウとの関連付け 65
- リスト 5-3 新しいタイプによるファイルの保存 66
- リスト 5-4 画像ファイルタイプのフィルタリング 67
- リスト 5-5 アクセサリビューを含むnibファイルのロード 70

ファイルとディレクトリの管理 72

- リスト 6-1 アプリケーションファイルのカスタムディレクトリの作成 73
- リスト 6-2 ディレクトリの非同期コピー 75

ファイルコーデインータを使用せずにファイルを読み書きする方法 78

- リスト 7-1 ディスパッチI/Oチャネルの作成 80
- リスト 7-2 ディスパッチI/Oチャネルを使用したテキストファイルからのバイトの読み取り 83

リスト 7-3 NSFileHandleを使用したファイルの内容の読み取り 86

リスト 7-4 POSIX関数を使用したファイルの内容の読み取り 87

OS X Libraryディレクトリの詳細 98

表 A-1 Libraryディレクトリのサブディレクトリ 98

ファイルシステムの詳細 102

図 B-1 権限の伝達 112

図 B-2 所有権と権限の情報 117

表 B-1 OS Xでサポートされるファイルシステム 102

表 B-2 ACLを使用したファイル権限ビット 107

表 B-3 BSDのファイル権限ビット 115

表 B-4 ファイルシステムの特別な権限ビット 115

ファイルとディレクトリの概要

ファイルシステムは、どのオペレーティングシステムにおいても重要な部分を占めます。各ユーザがコンテンツを保管する場所であるためです。ファイルシステムの構成は、ユーザのファイルの検索を支援する上で重要な役割を果たします。またアプリケーションおよびシステム自体についても、ユーザのサポートに必要なリソースの検出とアクセスが、ファイルシステムの構成により効率化します。

このドキュメントは、OS X、iOS、およびiCloud向けのソフトウェアを作成するデベロッパを対象としています。ファイルとディレクトリへのアクセスと、iCloud間でのファイルの移動にシステムインターフェイスを使用する方法を示し、ファイルの作業を最適化する方法をガイダンスします。また新規に作成したファイルを配置する場所についても説明します。

Important OS XアプリケーションにApp Sandboxを採用すると、ファイルシステムの機能の多くは、その動作が変わります。たとえば、アプリケーションのコンテナディレクトリ外にアクセスするためには、適切なエンタイトルメントを要求しなければなりません。ファイルシステム資源に対する永続的なアクセスを取得するためには、NSURLクラスまたはCFURLRef不透過型の、セキュリティ保護機能を組み込んだブックマーク機能を使う必要があります。アプリケーションの補助ファイルの位置が、ユーザのホームフォルダではなくコンテナを基準とした位置に変わります。また、「開く」、「保存」ダイアログも、AppKitではなくPowwerboxが提供するものになります。変更点の詳細については、『*App Sandbox Design Guide*』を参照してください。

At a Glance

ファイルシステムを効果的に活用するには、ファイルシステムから何を期待できるか、またファイルシステムのアクセスにどのような技術が利用できるかを把握する必要があります。

ファイルシステムの編成は特定化される

iOSとOS Xのファイルシステムは、ユーザとアプリケーションのいずれに対しても、ファイルの整合性が効率的に維持されるように構造化されています。コードレベルでは、システムのファイルシステムが正しく編成されていれば、アプリケーションで必要になるファイルは簡単に検索できます。言うまでもありませんが、作成したファイルを保存する場所を把握しておく必要があります。

関連する章と付録 [“ファイルシステムの基礎”](#)（12 ページ）、[“OS X Libraryディレクトリの詳細”](#)（98 ページ）、[“ファイルシステムの詳細”](#)（102 ページ）

ファイルへの安全なアクセス

OS Xなどのマルチユーザシステムでは、別のアプリケーションが使用中のファイルを、複数のアプリケーションが同時に読み書きを試みる場合があります。NSFileCoordinatorクラスとNSFilePresenterクラスは、ファイルの整合性の維持を可能にし、ファイルが別のアプリケーションで利用可能な場合（テキストエディットの最新ドキュメントをメールするなど）、最新バージョンのドキュメントの送信を保証します。

関連する章 [“ファイルコーディネータとファイルプレゼンタの役割”](#)（48 ページ）

ファイルへのアクセス方法はファイルタイプに依存

ファイルごとに、コードによる異なる処理が要求されます。アプリケーションで定義されるファイル形式の場合、バイナリストリームのバイトとしてコンテンツを読み取ることができます。一般的なファイル形式については、iOSとOS Xではファイルの読み書きを容易にする高度なサポートに対応しています。

関連するセクション [“ファイルアクセスの正しい方法の選択”](#)（31 ページ）

システムインターフェイスがファイルの検出と管理をサポート

ハードコードされたパス名は脆弱で、時間の経過とともに破損しやすくなります。このため周知の場所でファイルを検索するインターフェイスが、システムから提供されます。これらのインターフェイスの利用により、コードの堅牢性が強化され、互換性が維持され、その結果ファイルの場所に関わらず、ファイルを確実に見つけられるようになります。

関連する章 [“ファイルとディレクトリへのアクセス”](#)（31 ページ）

ユーザは標準のシステムパネルを使用してファイルと対話

ユーザが作成し管理するファイルについては、コードは標準の「Open」パネルと「Save」パネルを使用してファイルの場所を照会します。標準のパネルには、Finderの表示内容に一致する、ナビゲーション版のファイルシステムが表示されます。これらのパネルはカスタマイズせずに使用できますが、デフォルトの動作を修正したり、ユーザ独自のカスタムコンテンツで拡張することも可能です。サンド

ボックス内のアプリケーションも、パネルを使用してファイルにアクセスします。これはアプリケーションが下位のセキュリティレイヤと連動し、ユーザが明示的に選択したサンドボックス外部のファイルに対して、例外を許可するためです。

関連する章 [“OpenおよびSaveパネルの使用”](#) (62 ページ)

ファイルの非同期の読み書き

ファイル操作にはディスクまたはネットワークサーバへのアクセスが要求されるため、必ずアプリケーションの二次スレッドからファイルにアクセスすることになります。ファイルを読み書きする技術には、何も介入せずに非同期で実行されるものもありますが、独自の実行コンテキストの指定が要求されるものもあります。いずれの技術も本質的には同じものですが、簡易性と柔軟性のレベルが異なります。

ファイルを読み書きする場合、ファイルのアクションにより、同じファイルを使用するほかのアプリケーションに問題が生じるのを確実に防ぐために、ファイルコーディネータを使用する必要があります。

関連する章 [“ファイルコーディネータを使用せずにファイルを読み書きする方法”](#) (78 ページ)

Finderと同様のファイルの移動、コピー、削除、管理

システムインターフェイスは、Finderでサポートされるのと同じ種類の動作や、その他多くの動作をサポートします。ファイルとディレクトリを移動、コピー、作成、削除する操作は、ユーザと同様に実行できます。プログラムインターフェイスを使用すると、ディレクトリのコンテンツを反復したり、非表示のファイルをより効率的に操作したりできます。重要な点として、ほとんどの作業を非同期で実行できるため、アプリケーションのメインスレッドのブロックを回避できます。

関連する章 [“ファイルとディレクトリの管理”](#) (72 ページ)

ファイル関連の操作の最適化

ファイルシステムは、コンピュータでもっとも速度の遅い箇所のひとつであるため、効率的なコードの作成が重要になります。ファイルシステムコードの最適化とは、作業を最小限に抑え、実行する操作が効率的に行われることを保証することです。

関連する章 [“パフォーマンスのヒント”](#) (93 ページ)

関連項目

実行可能な高度なファイルシステムの作業の詳細については、『*File System Advanced Programming Topics*』を参照してください。

ファイルシステムの基礎

OS XとiOSのファイルシステムは、データファイルとアプリケーションの永続ストレージ、およびオペレーティングシステム自体に関連するファイル进行处理します。したがって、ファイルシステムはすべてのプロセスで使用される基本リソースのひとつと言えます。

OS XとiOSのファイルシステムは、いずれもUNIXファイルシステムをベースにしています。コンピュータに接続するすべてのディスクは、物理的にコンピュータにプラグインしているディスクや、ネットワークを経由して間接的に接続しているディスクもすべて含めて、1つのファイルコレクションを作成するためのスペースを提供します。ファイルの数はすぐに数百万規模に増加するため、ファイルシステムはディレクトリを使って階層的な組織を編成します。iOSとOS Xの基本的なディレクトリ構造は似ていますが、各システムでアプリケーションとユーザデータが組織化される方式が異なります。

ファイルシステムと相互作用するコードの作成を開始する前に、まずファイルシステムの編成の概要と、作成するコードに適用されるルールを理解する必要があります。適切なセキュリティ権限を持たないディレクトリへのファイルの書き込みを禁止する基本原則以外に、アプリケーションは正しく振る舞い、適切な場所にファイルを配置することが期待されます。ファイルを保存する正確な場所はプラットフォームに依存しますが、全般的な目標として、ユーザのファイルが常に検出しやすい状態に維持され、コードで使用するファイルがシステム内のユーザのアクセスできない場所に保管される必要があります。

iOSのファイルシステムについて

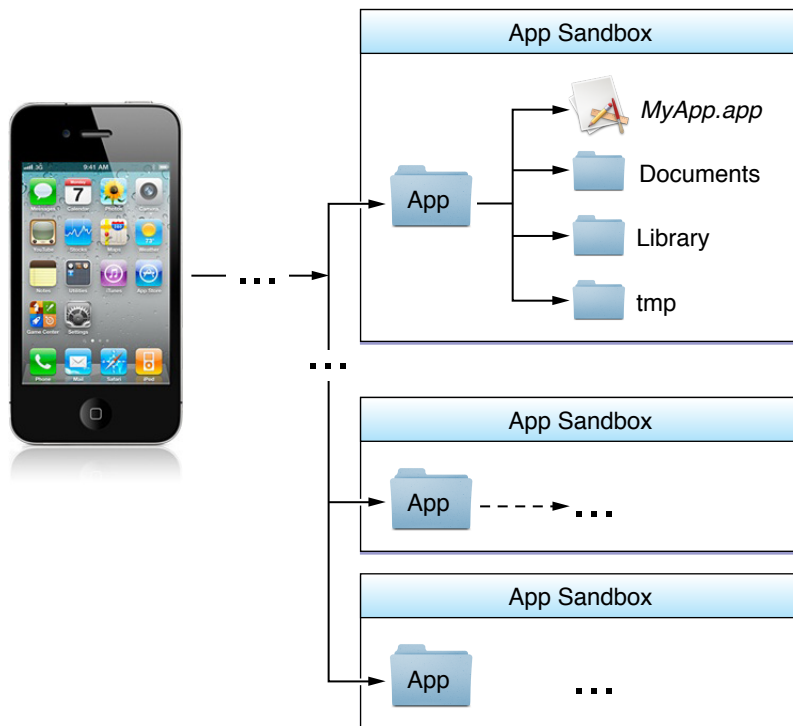
iOSのファイルシステムは、単独で実行されるアプリケーションに合わせて調整されています。システムの簡潔さを維持するために、iOSデバイスのユーザはファイルシステムに直接アクセスできず、アプリケーションもこの規則に従うことが要求されます。

島としてのアプリケーション

iOSアプリケーションのファイルシステムとの相互作用は、ほとんどがアプリケーションのサンドボックス内部のディレクトリに制限されます。新しいアプリケーションのインストール時に、インストーラコードによりアプリケーションのホームディレクトリが生成され、そのディレクトリにアプリケー

ションが配置され、その他の主要な複数のディレクトリが生成されます。これらのディレクトリは、ファイルシステムにおいてはアプリケーションのプライマリビューと呼ばれます。図 1-1に、アプリケーションのサンドボックスを図示しています。

図 1-1 各iOSアプリケーションは専用のサンドボックス内で動作



アプリケーションはサンドボックス内に置かれているため、一般にホームディレクトリの外部のディレクトリ内のファイルへのアクセス、またはそのようなファイルの作成が禁止されます。このルールの例外が起こるのは、アプリケーションがパブリックシステムインターフェイスを使用して、ユーザの連絡先や音楽などのコンテンツにアクセスする場合です。そのような場合、システムのフレームワークは、適切なデータストアからの読み取りやデータストアの修正に必要な、ファイルに関連したすべての操作を処理します。

アプリケーションのサンドボックス内のディレクトリの検索の詳細については、[“iOS標準ディレクトリ：ファイルの保存場所”](#)（13 ページ）を参照してください。

iOS標準ディレクトリ：ファイルの保存場所

セキュリティ上の理由により、iOSアプリケーションはデータを書き込む場所が制限されます。デバイスにアプリケーションがインストールされると、iTunesではこのアプリケーション用のホームディレクトリが作成されます。このディレクトリには、そのアプリケーションの領域が表示され、アプリケーションから直接アクセスできるすべてのコンテンツを含めます。表 1-1に、このホームディレ

クトリの重要なサブディレクトリの一部を示し、それぞれの用途を説明しています。またこの表には、各サブディレクトリの追加的なアクセス制限が記述され、ディレクトリの内容がiTunesによってバックアップされるかどうかを示されています。

表 1-1 iOSアプリケーションの共通に使用されるディレクトリ

ディレクトリ	解説
<Application_Home> / AppName .app	<p>これはアプリケーション自体を含む、バンドルのディレクトリです。このディレクトリには何も書き込まないでください。改ざんを防止するために、バンドルディレクトリはインストール時に署名されます。このディレクトリに書き込みをすると署名が変化するため、アプリケーションを再度起動することができなくなります。</p> <p>iOS 2.1以降では、このディレクトリの内容がiTunesによってバックアップされません。ただし、App Storeから購入したアプリケーションの最初の同期はiTunesによって実行されます。</p>
<Application_Home> /Documents/	<p>このディレクトリは、重要なユーザドキュメントやアプリケーションデータファイルを保存する場合に使用します。重要なデータとは、ユーザが作成した情報で、アプリケーションで生成し直すことができないもののことです。</p> <p>このディレクトリの内容は、ファイル共有によりユーザからのアクセスが可能になります。このディレクトリの内容はiTunesによってバックアップされます。</p>
<Application_Home> /Documents/Inbox	<p>このディレクトリは、外部エンティティからアプリケーションに対して要求されたファイルのアクセスに使用します。特に、「メール(Mail)」プログラムは、アプリケーションに関連した電子メール添付ファイルをこのディレクトリに保存します。Document Interaction Controllersもファイルを同じディレクトリに配置する場合があります。</p> <p>アプリケーションはこのディレクトリのファイルの読み取りと削除が可能です。新しいファイルの作成または既存のファイルへの書き込みは実行できません。ユーザがこのディレクトリのファイルの編集を試みた場合、アプリケーションが暗黙的にディレクトリからファイルを移動しなければ変更を行うことができません。</p> <p>このディレクトリの内容はiTunesによってバックアップされます。</p>

ディレクトリ	解説
<Application_Home> /Library/	<p>このディレクトリは、ユーザのデータファイル以外のファイル用の最上位ディレクトリです。通常、いくつかの標準的なサブディレクトリにファイルを保存しますが、ユーザに公開しないバックアップファイル用にカスタムサブディレクトリを作成することもできますこのディレクトリは、ユーザのデータファイル用には使用しないでください。</p> <p>このディレクトリの内容はiTunesによってバックアップされます（ただし、Cachesサブディレクトリは除く）。</p> <p>Libraryディレクトリの詳細については、“Libraryディレクトリはアプリケーション固有のファイルを保存”（23 ページ）を参照してください。</p>
<Application_Home> /tmp/	<p>このディレクトリは、アプリケーションを次に起動するまで保持する必要のない一時ファイルを書き込むために使用します。アプリケーションは、これらのファイルが不要になったと判断したら、このディレクトリから削除する必要があります（アプリケーションが実行されていないときに、システムが古いファイルをこのディレクトリから削除する場合もあります）。</p> <p>iOS 2.1以降では、このディレクトリの内容がiTunesによってバックアップされません。</p>

iOSアプリケーションは、Documents、Library、tmpなどの下に、追加のディレクトリを作成できます。これはファイルを整理して配置するために役立ちます。

上記のiOSアプリケーションのディレクトリの参照方法の詳細については、“[標準ディレクトリの項目の検索](#)”（37 ページ）を参照してください。ファイルの置き場所に関して、“[アプリケーションファイルの置き場所](#)”（15 ページ）にいくつかヒントを載せてあります。

アプリケーションファイルの置き場所

iOSデバイスの同期やバックアップに長期間かかるのを防ぐには、アプリケーションのホームディレクトリ内のファイルの置き場所を注意深く選択しておく必要があります。大容量のファイルを扱うアプリケーションの場合、iTunesやiCloudへのバックアップには相応の時間がかかります。もちろんストレージ容量を圧迫することにもなるので、積極的に削除して空き容量を増やす、あるいはiCloudへのバックアップを無効にする、という手段をユーザは考えてしまうでしょう。このことを念頭に、アプリケーションデータの保存に関しては、次のガイドラインに従って実装してください。

- ユーザデータは<Application_Home> /Documents/以下に置きます。ユーザデータとは、ユーザドキュメントなどユーザが作成した情報で、アプリケーションで生成し直すことができないもののことです。
- サポートファイル（アプリケーションがダウンロードまたは生成するファイルで、必要ならば再生成が可能なもの）は、次のいずれかの方法で管理します。

- iOS 5.0までは、サポートファイルは<Application_Home>/Library/Cachesディレクトリに置いて、バックアップの対象にならないようにしていました。
- iOS 5.0.1以降、<Application_Home>/Library/Application Supportディレクトリに置き、com.apple.MobileBackupという拡張属性を適用するようになっています。その結果、iTunesやiCloudへのバックアップ対象から外れます。大量のサポートファイルがある場合は、専用のサブディレクトリに格納し、このディレクトリにのみ拡張属性を適用しても構いません。
- データキャッシュファイルは<Application_Home>/Library/Cachesディレクトリ以下に置きます。このディレクトリに置くべきファイルとしては、データベースキャッシュファイルや、いつでもダウンロードし直せる雑誌、新聞、地図などがあります。アプリケーションは、システムが空きディスク空間を増やすためにキャッシュしたデータを削除しても、適切に対処できるようにしておかなければなりません。
- 一時データは<Application_Home>/tmpディレクトリに置いてください。一時データとは、長期間にわたって保存しておく必要がないデータのことです。使い終わったら削除して、デバイス上の空間を消費し続けないようにしなければなりません。

OS Xファイルシステムの概要

OS Xファイルシステムは、ユーザとソフトウェアがいずれもファイルシステムにアクセスする、Macintoshコンピュータ向けに設計されています。ユーザはFinderから直接ファイルシステムにアクセスします。Finderは一部のファイルとディレクトリを非表示あるいは別名で表示し、ファイルシステムをユーザ指向のビュー形式で示します。アプリケーションは、ディスクに表示されときの完全なファイルシステムを示すシステムインターフェイスを使用して、ファイルシステムにアクセスします。

ファイルの配置はドメインに依存

OSXでは、ファイルシステムは複数のドメインに分割されます。ドメイン内ではそれぞれの用途に基づいて、ファイルとリソースが分類されます。この分類により、ユーザはファイルの特定のサブセットにのみ配慮すればよいため、ファイル操作が簡潔になります。またファイルのドメイン別の配列により、ドメイン内のファイルへの一括アクセス権が適用され、不正なユーザによる意図的あるいは不注意なファイルの変更を防ぐことができます。

ユーザドメインは、システムにログインしたユーザに固有のリソースを含みます。このドメインは理論的にはすべてのユーザを対象としますが、実行時のユーザのホームディレクトリのみを反映します。ユーザのホームディレクトリは、コンピュータのブートボリューム（/Usersディレクトリ）またはネットワークボリューム上に置くことができます。各ユーザは（権限に関係なく）、各自のホームディレクトリにアクセスし、同ディレクトリを制御することができます。

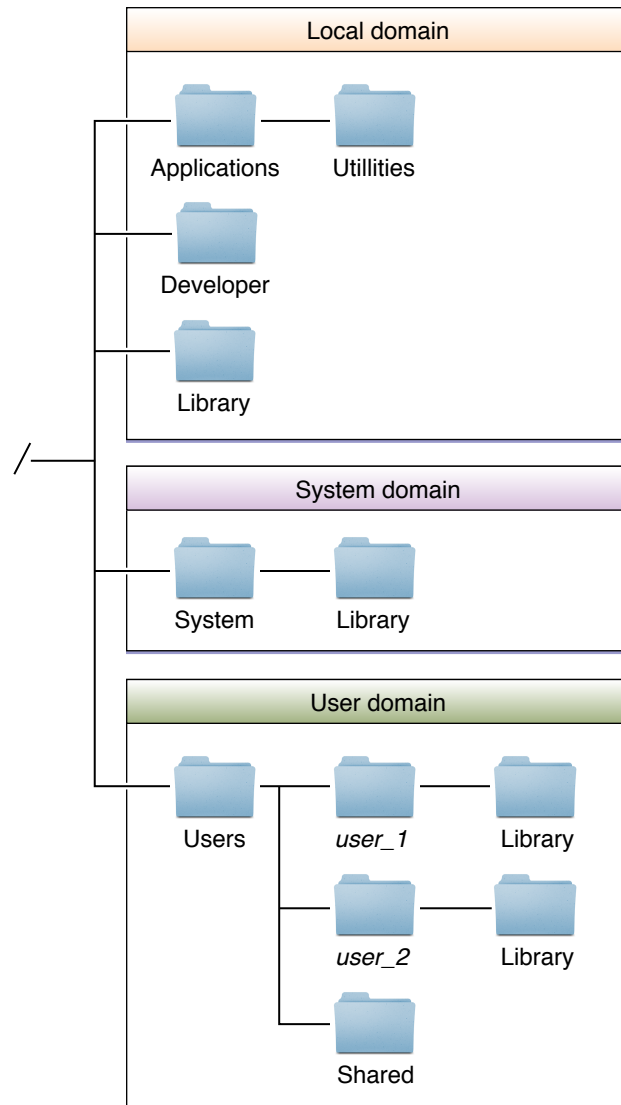
ローカルドメインは、現在のコンピュータのローカルアプリケーションなどのリソース、および現在のコンピュータの全ユーザ間で共有されるリソースを含みます。ローカルドメインは、単一の物理ディレクトリに対応しませんが、ローカルのブート（およびルート）ボリュームの複数のディレクトリから構成されます。このドメインは通常はシステムにより管理されますが、管理権限を持つユーザもこのドメインの項目の追加、削除、または修正が可能です。

ネットワークドメインは、ローカルエリアネットワーク（LAN）の全ユーザ間で共有される、アプリケーションとドキュメントなどのリソースを含みます。このドメインの項目は、通常はネットワークファイルサーバ上に配置され、ネットワーク管理者の管理下に置かれます。

システムドメインは、Appleでインストールされるシステムソフトウェアを含みます。システムドメインのリソースは、システムの実行時に要求されます。ユーザはこのドメインの項目の追加、削除、または変更が禁止されています。

図 1-2に、OS Xインストール時のローカルファイルシステムへの、Local、System、Userの各ドメインのマッピングプロセスを示しています（ネットワークドメインは示されていませんが、多くの点でローカルドメインに類似しています）。この図には、ユーザーに表示されるディレクトリが示されています。ユーザーのシステムに応じて、ほかのディレクトリが表示される場合や、図に示されたディレクトリが表示されない場合があります。

図 1-2 ローカルOS Xファイルシステム



OS Xのディレクトリの内容の詳細については、“[OS X標準ディレクトリ：ファイルの保存場所](#)”（19ページ）を参照してください。OS Xで通常はユーザーに表示されないディレクトリの詳細（およびその理由）については、“[非表示ファイルとディレクトリ：ユーザー体験の簡素化](#)”（21ページ）を参照してください。

OS X標準ディレクトリ：ファイルの保存場所

システム供給かアプリケーションで生成されるかを問わず、すべてのファイルはOS X内の場所が割り当てられます。表 1-2に、OS Xインストール時の最上位ディレクトリの一部と、各ディレクトリに含まれるコンテンツのタイプを一覧にしています。

表 1-2 OS Xの共通に使用されるディレクトリ

ディレクトリ	使用方法
/Applications	<p>このディレクトリに、コンピュータの全ユーザに使用されるアプリケーションをインストールします。App Storeは、ユーザが購入したアプリケーションを自動的にこのディレクトリにインストールします。</p> <p>Utilitiesサブディレクトリは、ローカルシステムの管理に使用されるアプリケーションのサブセットを含みます。</p> <p>このディレクトリは、ローカルドメインに含まれます。</p>
/Developer	<p>このディレクトリは、Xcodeアプリケーション、ツール、デベロッパ関連のリソースを含みます。</p> <p>このディレクトリのデフォルト名はDeveloperですが、ユーザが変更することもできます。複数のXcodeツールをインストールするために、名前を変更することはよくあります。名前とは無関係に、このディレクトリはローカルドメインに含まれます。</p>
Library	<p>システムには複数のLibraryディレクトリが存在し、それぞれが異なるドメインまたは特定のユーザに関連付けられています。アプリケーションは、アプリケーション固有（またはシステム固有）のリソースの保存にLibraryディレクトリを使用します。</p> <p>このディレクトリの内容の詳細、およびこのディレクトリを使用してアプリケーションをサポートする方法の詳細については、“Libraryディレクトリはアプリケーション固有のファイルを保存”（23 ページ）を参照してください。</p>
/Network	<p>このディレクトリは、ローカルエリアネットワーク（LAN）のコンピュータのリストを含みます。</p> <p>ネットワークファイルサーバ上のファイルのパスは、/Networkディレクトリで始まるとは限りません。パス名は、ネットワークのマウント量など複数の要因に応じて変わります。たとえば、ユーザがConnect to Serverコマンドを使用してボリュームをマウントする場合、パスは/Volumesディレクトリから開始します。コードを記述する場合、ブートボリューム以外のすべてのボリュームのファイルを、ネットワークベースのサーバ上に配置できると仮定してください。</p>

ディレクトリ	使用方法
/System	<p>このディレクトリは、OS Xの実行に必要とされるシステムリソースを含みます。これらのリソースはAppleにより供給され、変更できません。</p> <p>このディレクトリには、システムドメインの内容が含まれます。</p>
/Users	<p>このディレクトリは、1つまたは複数のユーザホームディレクトリを含みます。ユーザホームディレクトリは、ユーザ関連のファイルが保存される場所です。通常のユーザのホームディレクトリには、以下のサブディレクトリが含まれます。</p> <ul style="list-style-type: none">Applications—ユーザ固有のアプリケーションを含みます。Desktop—ユーザのデスクトップの項目を含みます。Documents—ユーザのドキュメントとファイルを含みます。Downloads—インターネットからダウンロードされたファイルを含みます。Library—ユーザ固有のアプリケーションファイルを含みます（OS X 10.7以降では非表示）。Movies—ユーザのビデオファイルを含みます。Music—ユーザの音楽ファイルを含みます。Pictures—ユーザの写真を含みます。Public—ユーザが共有を予定しているコンテンツを含みます。Sites—ユーザの個人サイトで使用されるウェブページを含みます（これらのページを表示する場合、Web共有を有効にする必要があります）。 <p>上記のディレクトリは、ユーザのドキュメントとメディアの保存専用に使 用されます。アプリケーションは、ユーザにより明示的に指示されてい る場合を除き、上記のディレクトリにファイルを書き込むことができま せん。このルールの一例外はLibraryディレクトリです。このディレク トリは、現在のユーザのサポートに必要なデータファイルの保存にアプリ ケーションで使われる場合があります。</p> <p>サブディレクトリの中でPublicディレクトリのみが、システムのほかの ユーザからアクセスできます。ほかのディレクトリへのアクセスは、デ フォルトで制限されます。</p>

Important ユーザのDocumentsとDesktopディレクトリ内のファイルは、ユーザが作成し、ユーザが直接操作するドキュメントのみを反映します。同様に、メディアディレクトリはユーザのメディアファイルのみを含みます。前記のディレクトリは、アプリケーションで自動的に生成され、管理されるデータファイルの保存に使用できません。自動的に生成されたファイルを保存する場所が必要になる場合は、この目的のために個別に指定されているLibraryディレクトリを使用します。Libraryディレクトリのファイルを保存する場所の詳細については、“[Libraryディレクトリはアプリケーション固有のファイルを保存](#)”（23 ページ）を参照してください。

表 1-2（19 ページ）のディレクトリは、OS Xユーザに表示されるディレクトリですが、ファイルシステムに存在する唯一のディレクトリではありません。OS Xは多くのディレクトリを非表示にし、ユーザによる必要のないファイルアクセスを防ぎます。

サンドボックス内のOS Xアプリケーションファイルのコンテナ

サンドボックス内のOS Xアプリケーションは、それぞれのApplication Support、Cache、一時ディレクトリ、その他の関連ドキュメントをすべて、システム定義のパス（NSHomeDirectory関数で取得可）で示されるディレクトリ内に保存します。

詳細については、『*App Sandbox Design Guide*』を参照してください。

非表示ファイルとディレクトリ：ユーザ体験の簡素化

ユーザ体験を簡素化するために、Finderとユーザが使用する一部の特定のインターフェイス（OpenパネルとSaveパネルなど）は、ユーザが使用する必要のない多くのファイルとディレクトリを非表示にしています。非表示項目の多くは、ユーザが直接アクセスできない（あるいはアクセスすべきではない）システム固有またはアプリケーション固有のリソースです。表示されないファイルとディレクトリには以下のようなものがあります。

- **Dotディレクトリとファイル**—名前がピリオド（.）文字で始まるファイルまたはディレクトリは、自動的に非表示になります。これはUNIXの規約に基づき、システムスクリプトとその他の特殊なタイプのファイルとディレクトリの非表示に使用されていました。このカテゴリの2つの特別なディレクトリは、それぞれ現在のディレクトリと親ディレクトリを参照する、.と..ディレクトリです。
- **UNIX固有のディレクトリ**—このカテゴリのディレクトリは、従来のUNIXのインストールから継承されています。システムのBSDレイヤの重要な部分ですが、エンドユーザ以上にソフトウェアデベロッパーにとって有用です。非表示の重要なディレクトリの一部を以下に示します。
 - /bin—必須のコマンドラインバイナリを含みます。通常は、これらのバイナリはコマンドラインスクリプトから実行します。
 - /dev—接続されたハードウェアのマウント箇所など、必須デバイスファイルを含みます。
 - /etc—ホスト固有の設定ファイルを含みます。

- `/sbin`—必須のシステムバイナリを含みます。
- `/tmp`—アプリケーションとシステムで生成された一時ファイルを含みます。
- `/usr`—必須ではないコマンドラインバイナリ、ライブラリ、ヘッダファイル、その他のデータを含みます。
- `/var`—ログファイルと、その他のさまざまな内容のファイルを含みます（ログファイルは通常は、コンソールアプリケーションで表示されます）。
- **明示的に非表示にされたファイルとディレクトリ**—Finderはユーザが直接アクセスできない特定のファイルまたはディレクトリを、非表示にする場合があります。このもっとも特筆すべき例として`/Volumes`ディレクトリがあり、ここにはコマンドラインからローカルファイルシステムにマウントされた、各ディスクのサブディレクトリが含まれます（Finderはローカルディスクのアクセス用に、別のユーザインターフェイスを提供しています）。OS X 10.7以降では、Finderは`~/Library`ディレクトリ、すなわちユーザのホームディレクトリ内の`Library`ディレクトリも非表示にします。
- **パッケージとバンドル**—パッケージとバンドルは、Finderがファイルと同様にユーザに提示するディレクトリです。バンドルはアプリケーションなどの実行可能ファイルの内部作業を隠し、ファイルシステムの外部での移動が可能な単一のエンティティのみを表示します。同様に、パッケージにより個別の複数のファイルから構成される複雑なドキュメント形式の実装がアプリケーションで可能になり、同時に、ユーザに対して表面上は単一ドキュメントに見られる内容を表示し続けることができます。

Finderとその他のシステムインターフェイスは、ファイルとディレクトリをユーザに非表示にしますが、`NSFileManager`などのCocoaインターフェイスは、通常はユーザに表示されないファイルまたはディレクトリをフィルタリングしません。このため、これらのインターフェイスを使用するコードは、理論的には、ファイルシステムとその内容を完全に収めたビューを保持しています（実際には、プロセスがアクセスするのは、適切な権限を持つファイルとディレクトリに限定されます）。

ファイルとディレクトリに代替名を割り当てられる

状況によっては、Finderはファイルシステム内の実際の名称と一致しないファイル名またはディレクトリ名をユーザに提示します。これらの名前は**表示名**として知られ、ファイルとディレクトリの情報をユーザに提示するときに、Finderと特定のシステムコンポーネント（`Open`パネルと`Save`パネルなど）によってのみ使用されます。表示名は内容をより分かりやすくユーザに示すため、ユーザ体験の向上が図られます。たとえば、OS Xは以下のような状況で表示名を使用します。

- **ローカライズ名**—多くのシステムディレクトリに対して、`Applications`、`Library`、`Music`、`Movies`などのローカライズ名が提供されます。アプリケーションも同様に、アプリケーション自体およびアプリケーションで作成されたディレクトリにローカライズ名を割り当てます。

- **ファイル名拡張子の非表示**—デフォルトで、すべてのファイルのファイル名拡張子は非表示です。ユーザはオプションを変更できますが、ファイル名拡張子の非表示が有効な場合、ファイル名の最後のピリオドの後の文字（およびピリオド）は表示されません。

表示名は、ファイルシステム内のファイルの実際の名称を反映しません。プログラムによりファイルまたはディレクトリにアクセスするコードは、ファイルシステムのインターフェイスを使用して項目を開いたり操作するときの、項目の実際の名前を指定する必要があります。アプリケーションで表示名が使用されるのは、ファイルまたはディレクトリの名前をユーザに表示する場合に限定されます。NSFileManagerのdisplayNameAtPath:メソッドを使用すると、任意のファイルまたはディレクトリの表示名を取得できます。

Important コードを使用して、表示名の直接の修正をユーザに許可することはできません。ファイルの名前をユーザ側で指定する必要がある場合は、Saveパネルを使用します。

アプリケーションで生成されるディレクトリのローカライズ方法の詳細については、『*File System Advanced Programming Topics*』を参照してください。アプリケーションの内容のローカライズの詳細については、『*Internationalization Programming Topics*』を参照してください。

Libraryディレクトリはアプリケーション固有のファイルを保存

Libraryディレクトリは、アプリケーションとその他のコードモジュールがカスタムデータファイルを保存する場所です。iOSとOS Xのいずれのコードを記述する場合でも、Libraryディレクトリの構造を理解することは重要です。このディレクトリは、データファイル、キャッシュ、リソース、環境設定、また特定の状況ではユーザデータの保存にも使用します。

システム全体に複数のLibraryディレクトリがありますが、コードからアクセスが必要になるのは一部に過ぎません。

- 現在のホームディレクトリのLibrary—ユーザ固有のすべてのファイルが格納されるため、ディレクトリのもっとも多く利用されるバージョンです。iOSでは、ホームディレクトリはアプリケーションのサンドボックスディレクトリです。OS Xでは、アプリケーションのサンドボックスディレクトリ、または（アプリケーションがサンドボックス内にインストールされていない場合は）現在のユーザのホームディレクトリです。
- /Library (OS Xのみ) —ユーザ間でリソースを共有するアプリケーションは、リソースをこのバージョンのLibraryディレクトリに保存します。サンドボックス内のアプリケーションは、このディレクトリの使用が許可されていません。
- /System/Library (OS Xのみ) —このディレクトリはAppleで予約されています。

使用するLibraryディレクトリのバージョンを選択した後、ファイルを保存する場所を確認する必要があります。Libraryディレクトリ自体には、アプリケーション固有の内容がいくつかの一般的なカテゴリに細分化される、複数のサブディレクトリが含まれます。表 1-3に、もっとも共通に使用されるサブディレクトリをリストアップしています。OS XのLibraryディレクトリには、リスト内に示されていない多くのサブディレクトリが含まれますが、ほとんどは用途がシステムに限定されます。サブディレクトリの詳細なリストについては、“[OS X Libraryディレクトリの詳細](#)”（98 ページ）を参照してください。

表 1-3 Libraryディレクトリの主要なサブディレクトリ

ディレクトリ	使用方法
Application Support	<p>このディレクトリは、ユーザのドキュメントに関連付けられていないアプリケーションデータファイルを保存する場合に使用します。たとえば、アプリケーションで生成されたデータファイル、設定ファイル、テンプレート、その他のアプリケーションで管理される固定リソースまたは修正可能なリソースの保存にこのディレクトリを使用できます。アプリケーションも、アプリケーションのバンドルに元々格納されていたリソースの修正可能なコピーを保存する場合に、このディレクトリを使用する場合があります。ゲームでは、ユーザが購入した新しいレベル、またサーバからダウンロードされた新しいレベルの保存にこのディレクトリが使用されます。</p> <p>このディレクトリのすべての内容は、アプリケーションのバンドルIDまたは社名を名前に使った、カスタムサブディレクトリに置かれます。</p> <p>iOSでは、このディレクトリの内容はiTunesによってバックアップされます。</p>
Caches	<p>このディレクトリは、アプリケーションが簡単に作成し直すことが可能な、アプリケーション固有のサポートファイルの記述に使用します。一般的にアプリケーションはこのディレクトリの内容の管理、および必要に応じたファイルの追加と削除に責任を持ちます。</p> <p>iOS 2.2以降では、このディレクトリの内容がiTunesによってバックアップされません。また、iTunesはデバイスの完全な復元の際に、このディレクトリのファイルを削除します。</p> <p>iOS 5.0以降では、まれですが、システムのディスクスペースが非常に少ない場合に、Cachesディレクトリが削除される場合があります。このような削除は、アプリケーションの実行中は起こりません。ただし、Cachesディレクトリが消去される状況は、iTunesの復元時に限らないことを覚えておいてください。</p>

ディレクトリ	使用方法
Frameworks	OS Xでは、複数のアプリケーションで共有されるフレームワークは、ローカルまたはユーザドメインのいずれかにインストールできます。システムドメインのFrameworksディレクトリは、OS Xアプリケーションの作成時に使用するフレームワークを保存します。 iOSでは、アプリケーションはカスタムフレームワークをインストールできません。
Preferences	このディレクトリには、アプリケーション固有の環境設定ファイルを保存します。ディレクトリ内に、デベロッパ自身がファイルを作成してはいけません。デベロッパのファイルには、NSUserDefaultsクラスまたはCPreferences APIを使用して、アプリケーションの環境設定の値を取得し設定します。 iOSでは、このディレクトリの内容はiTunesによってバックアップされます。

iCloudファイルのストレージコンテナ

iCloudは、iCloudを利用するアプリケーションのファイルを保存するための構造化されたシステムを提供します。

- アプリケーションは、ネイティブファイルを保存するためのプライマリiCloudコンテナディレクトリを保有します。またアプリケーションのエントリーメントのリスト内の、セカンダリiCloudコンテナディレクトリにもアクセスできます。
- 各コンテナディレクトリ内部で、ファイルは「ドキュメント」とデータに区分されます。Documentsサブディレクトリ（またはそのサブディレクトリの1つ）内のすべてのファイルまたはファイルパッケージは、個別に削除できる独立したドキュメントとして（OS XとiOSのiCloud UIを通じて）ユーザに表示されます。Documentsまたはそのサブディレクトリのいずれかに存在しない項目は、データとして処理され、iCloud UIに単一のエン트리として表示されます。

ユーザがアプリケーションのユーザインターフェイスで作成し、表示するドキュメント（たとえば、Pages、Numbers、Keynoteから開いたり、保存したりするドキュメント）は、Documentsディレクトリに保存されます。保存済みのゲームもDocumentsディレクトリに保存されるファイルの例です。ゲームの保存時にも、アプリケーションからある種の選択方法が提供される可能性があるためです。

アプリケーションで、ユーザによる直接の表示または修正を禁止するものは、Documentsディレクトリの外部に配置する必要があります。アプリケーションはコンテナディレクトリ内部にサブディレクトリを作成できるため、必要に応じて、プライベートファイルを配列できます。

アプリケーションは、ローカルのファイルとディレクトリを作成するのとまったく同じ方法で、iCloud コンテナディレクトリにファイルとディレクトリを作成します。またすべてのファイルの属性が保存され、ファイルに拡張属性が追加されると、それらの属性はiCloudとユーザのほかのデバイスにもコピーされます。

またiCloudコンテナでは、ドキュメント形式を作成せずに簡単にアクセスできるキー値ペアを保存できます。

ファイルの内容のタイプの識別方法

ファイルの内容のタイプは、主に2種類の方法で識別します。

- Uniform Type Identifiers (UTI)
- ファイル名拡張子

Uniform Type Identifier (UTI) は、「タイプ」を持つと見なされたエンティティのクラスを一意に識別する文字列です。UTIはすべてのアプリケーションとサービスが認識し、信頼できる、一貫性のあるIDをデータに割り当てます。またUTIはほかの方法よりも柔軟性があります。これはファイルとディレクトリ以外に、データのタイプの表現にも使用できるためです。UTIの一例を示します。

`public.text`—テキストデータを識別するパブリックタイプ。

`public.jpeg`—JPEG画像データを識別するパブリックタイプ。

`com.apple.bundle`—バンドルディレクトリを識別するAppleのタイプ。

`com.apple.application-bundle`—バンドルアプリケーションを識別するAppleのタイプ。

UTIベースのインターフェイスがファイルタイプの指定に使用できる場合、必ずそのインターフェイスを優先的に使用してください。多くのOS Xインターフェイスでは、作業の必要なファイルまたはディレクトリに対応したUTIを指定できます。たとえば、**Open**パネルでファイルフィルタとしてUTIを使用し、ユーザが選択できるファイルのタイプをアプリケーションの処理が可能なタイプに制限できます。`NSDocument`、`NSPasteboard`、`NSImage`を含む複数のAppKitクラスがUTIをサポートしています。iOSでは、UTIはペーストボードのタイプの指定にのみ使用されます。

特定のファイルのUTIを判断する方法の1つに、ファイル名拡張子の確認があります。**ファイル名拡張子**は、ファイルの末尾に付加される文字列で、ピリオドでファイル主部から分離されます。固有の文字列はそれぞれ、特定のタイプのファイルを識別します。たとえば、`.strings`拡張子はローカライズ可能な文字列データのリソースファイルを識別しますが、`.png`拡張子は、移植可能なネットワークグラフィック形式の画像データのファイルを識別します。

注意 OS XとiOSファイル名ではピリオド文字が有効であるため、ファイル名の最後のピリオド後の文字のみがファイル名拡張子の部分と見なされます。最後のピリオドの左の部分は、ファイル名部分と見なされます。

アプリケーションでカスタムファイル形式が定義されている場合、それらの形式と関連するファイル名拡張子をアプリケーションのInfo.plistファイルに登録する必要があります。

CFBundleDocumentTypesキーは、アプリケーションで認識され、開くことができるファイル形式を指定します。カスタムファイル形式のエントリには、ファイル名拡張子、さらにファイルの内容に対応したUTIを含める必要があります。その情報に基づいて、該当するタイプのファイルがアプリケーションにダイレクトされます。

UTIおよびその使い方については、『*Uniform Type Identifiers Overview*』を参照してください。

CFBundleDocumentTypesキーの詳細については、『*Information Property List Key Reference*』を参照してください。

セキュリティ：作成するファイルの保護

すべてのユーザデータとシステムコードはディスクのどこかに保存されるため、ファイルの整合性とファイルシステムの保護は重要な作業です。そのような理由により、内容の保護、および内容の盗難やほかのプロセスによる破損の防止には複数の方法が使用されます。

ファイルの作業時のセキュアなコーディング方式の詳細については、『*Secure Coding Guide*』を参照してください。

サンドボックスは被害の拡大を制限

iOSとOS X v10.7以降では、アプリケーションによるファイルシステムの禁止箇所への書き込みは、サンドボックスにより回避されます。サンドボックス内の各アプリケーションは、ファイルを書き込むための専用のコンテナディレクトリが割り当てられます。アプリケーションはほかのアプリケーションのコンテナ、またはサンドボックスの外部のほとんどのディレクトリに書き込むことができません。これらの制約により、アプリケーションのセキュリティ違反が起きたときの、潜在的な被害が制限されます。

OS X v10.7以降のアプリケーションを作成するデベロッパは、セキュリティを強化するために、作業中のアプリケーションをサンドボックスに保存することが奨励されます。iOSアプリケーションのデベロッパは、iOSのインストール時にシステムで自動的に設定されるため、アプリケーションを明示的にサンドボックスに保存する必要はありません。

サンドボックスと、ファイルシステムのアクセスに対する制約のタイプの詳細については、『*Mac App Programming Guide*』と『*App Sandbox Design Guide*』を参照してください。

すべてのファイルアクセスに適用される許可とアクセス制御リスト

ファイルとディレクトリへのアクセスは、アクセス制御リスト（ACL）とBSD許可の組み合わせにより管理されます。アクセス制御リストは、ファイルまたはディレクトリに対して実行可能な内容と不可能な内容、および実行者を正確に定義する、きめの細かい制御リストです。アクセス制御リストに基づき、特定のファイルまたはディレクトリへの異なるレベルのアクセス権を、個々のユーザに付与することができます。これに対して、BSD許可はファイルの所有者、指定した特定のユーザグループ、全ユーザの3つのユーザクラスにのみアクセスを許可できます。詳細については、『*Security Overview*』を参照してください。

注意 ネットワークサーバのファイルについては、ファイルに関連したACLおよびBSD許可に関する前提は不要です。一部のネットワークファイルシステムは、この情報を要約版で提供します。

iOSアプリケーションは必ずサンドボックスで実行されるため、各アプリケーションで生成されたファイルに特定のACLと許可が割り当てられます。ただし、OS Xアプリケーションは、アクセスするファイルのアクセス制御リストの管理にIdentity Servicesを使用します。Identity Services（およびCollaborationフレームワーク）の使い方の詳細については、『*Identity Services Programming Guide*』を参照してください。

ディスクでのファイルの暗号化

OS XとiOSはいずれも、ディスクでのファイルの暗号化をサポートしています。

- iOS—iOSアプリケーションは、ディスクでの暗号化が必要なファイルを指定できます。ユーザが暗号化されたファイルを含むデバイスのロックを解除すると、アプリケーションの暗号化されたファイルへのアクセスを許可する復号キーが生成されます。ユーザがデバイスをロックすると、復号キーが破壊され、ファイルへの不正なアクセスが禁止されます。
- OS X—ユーザはディスクユーティリティアプリケーションを使用して、ボリュームの内容を暗号化できます（またシステム環境設定Security & Privacyから、ブートボリュームを暗号化することもできます）。暗号化されたディスクの内容をアプリケーションが利用できるのは、コンピュータの稼働時に限られます。ユーザがコンピュータをスリープに移行する、またはシャットダウンすると、復号キーが破壊され、ディスクの内容への不正なアクセスが回避されます。

iOSでは、ディスクベースの暗号化を利用するアプリケーションは、ユーザがデバイスをロックした場合、暗号化されたファイルの利用を停止する必要があります。デバイスのロックにより復号キーが破壊されるため、暗号化されたファイルへのアクセスはデバイスのアンロック時に制限されます。デバイスがロックされている間にiOSアプリケーションをバックグラウンドで実行できる場合、暗号化

されたファイルにアクセスせずにこの実行が可能でなければなりません。OSXの暗号化されたディスクは、コンピュータの稼働中は常にアクセスできるため、OSXアプリケーションはディスクレベルの暗号化の処理のために、特別な操作を行う必要がありません。

iOSで暗号化されたファイルを扱う方法の詳細については、『*iOS App Programming Guide*』を参照してください。

同期によりファイル関連のコードの強度を保証

ファイルシステムは、サードパーティ製アプリケーションとシステムアプリケーションで共有されるリソースです。複数のアプリケーションが同時にファイルとディレクトリにアクセスできるため、特定のアプリケーションの変更により、別のアプリケーションに古いファイルシステムが表示される可能性が生じます。別のアプリケーションでこのような変更の処理に対応できない場合、未知の状態あるいはクラッシュに陥る可能性があります。アプリケーションが特定のファイルの存在に依存している場合、そのようなファイルの変更の通知に同期化インターフェイスを使用できます。

ファイルシステムの同期化は、主にOSXで問題になります。OSXではユーザがFinderを直接操作する、またほかの任意の数のアプリケーションを同時に操作するためです。ただし、OSXでは同期化の問題に対処する以下のインターフェイスが提供されます。

- ファイルコーディネーター—OS X 10.7以降では、細かな同期化サポートをアプリケーションのオブジェクトに直接組み入れる手段として、ファイルコーディネータが提供されています。[“ファイルコーディネータとファイルプレゼンタの役割”](#)（48 ページ）を参照してください。
- FSEvents—OS X 10.5以降では、ファイルシステムのイベントを使って、ディレクトリまたはその内容の変更を監視することができます。『*File System Events Programming Guide*』を参照してください。

ファイル、並列処理、スレッドの安全性

ファイル関連の操作にはハードディスクとの相互作用が伴い、ほかの大半の操作よりも速度が遅くなるため、iOSとOSXのファイル関連の大半のインターフェイスは並列処理を前提に設計されています。設計に非同期操作を組み込む技術もありますが、ほとんどの技術はディスパッチキューまたは二次スレッドから安全に操作を実行できます。表1-4に、本書で説明する主要な技術の一部と、特定のスレッドまたは任意のスレッドから使用する場合の安全性を一覧にしています。インターフェイスの機能の詳細については、そのインターフェイスのリファレンスドキュメントを参照してください。

表 1-4 主要なクラスと技術のスレッドの安全性

クラス/技術	メモ
NSFileManager	ほとんどの作業においては、デフォルトのNSFileManagerオブジェクトを複数のバックグラウンドスレッドから同時に使用しても安全です。この規則の唯一の例外として、ファイルマネージャのデリゲートと相互作用する作業があげられます。ファイルマネージャオブジェクトをデリゲートで使用する場合、NSFileManagerクラスの一意的インスタンスを作成し、そのインスタンスをデリゲートで使うことが推奨されます。この一意的インスタンスは、1度に1つのスレッドから使用します。
Grand Central Dispatch	GCD自体はどのスレッドから使用しても安全です。ただし、ブロックをスレッドが安全な方法で記述する責任は作者の側にあります。
NSFileHandle、NSData、Cocoaストリーム	ファイルデータの読み書きに使用するFoundationオブジェクトの大半は、任意のシングルスレッドから使用できますが、複数のスレッドから同時に使用することはできません。
OpenパネルとSaveパネル	OpenパネルとSaveパネルはユーザインターフェイスの一部であるため、両パネルは必ずアプリケーションのメインスレッドから提示し、操作する必要があります。
POSIXルーチン	ファイルを操作するためのPOSIXルーチンは、一般に、任意のスレッドから安全に操作できるように設計されています。詳細については、対応するmanページを参照してください。
NSURLとNSString	パスの指定に使用する不変オブジェクトは、任意のスレッドから安全に使用できます。このオブジェクトは不変であるため、複数のスレッドから同時に参照できます。当然ですが、これらのオブジェクトの可変バージョンは、1度に1つのスレッドからのみ使用します。
NSEnumeratorとそのサブクラス	Enumeratorオブジェクトは、任意のシングルスレッドから安全に使用できますが、複数のスレッドから同時に使用できません。

ファイルの操作にスレッドセーフなインターフェイスを使用する場合においても、複数のスレッドまたは複数のプロセスで同じファイルの操作を試みると問題が生じます。複数のクライアントが同時に同じファイルを修正するのを禁止するセーフガードはとられていますが、そのようなセーフガードによりファイルへの排他的なアクセスが常に保証されるとは限りません（ほかのプロセスによる共有ファイルへのアクセスを禁止することもできません）。共有ファイルへの変更をコードに確実に認識させるためには、ファイルコーディネータを使用して、そのようなファイルへのアクセスを管理します。ファイルコーディネータの詳細については、“[ファイルコーディネータとファイルプレゼンタの役割](#)”（48 ページ）を参照してください。

ファイルとディレクトリへのアクセス

ファイルを開く前に、まずファイルシステムでファイルを検索する必要があります。システムフレームワークでは、Libraryディレクトリとその内容など、多くの一般的なディレクトリの参照を取得するためのルーチンが数多く提供されます。また既知のディレクトリ名からURLまたは文字列のパスを構築し、手動で場所を指定することもできます。

ファイルの場所が分かっている場合、ファイルアクセスの最適な方法のプランニングを始められます。ファイルのタイプに応じて、複数のオプションがあります。既知のファイルタイプの場合、通常は組み込みのシステムルーチンを使用して、ファイルの内容の読み書きと使用可能なオブジェクトの割り当てを行います。カスタムファイルタイプの場合、未処理のファイルデータを自分で読み取る必要が生じる場合があります。

ファイルアクセスの正しい方法の選択

バイトのストリームとして任意のファイルを開き、その内容を読み取ることができますが、必ずしも適切な選択肢とならない場合があります。OS XとiOSでは、組み込みサポートにより、多くのタイプの標準ファイル形式（テキストファイル、画像、サウンド、プロパティリストなど）を簡単に開くことができます。これらの標準ファイル形式については、ファイルの内容の読み書きに高レベルのオプションを使用します。表 2-1に、システムでサポートされる共通のファイルタイプと、アクセス方法に関する詳細を記述しています。

表 2-1 特別なルーチンを用いるファイルタイプ

ファイルタイプ	例	解説
リソースファイル	nibファイル 画像ファイル サウンドファイル 文字列ファイル ローカライズされたリソース	アプリケーションでは、アプリケーションを使用するコードとは無関係なデータの保存に、リソースファイルが使用されます。リソースファイルは、文字列と画像などのローカライズ可能な内容の保存に共通に使用されます。リソースファイルからデータを読み取るプロセスは、リソースのタイプに依存します。 リソースファイルの内容を読み取る方法については、『 <i>Resource Programming Guide</i> 』を参照してください。

ファイル タイプ	例	解説
テキスト ファイル	プレーンテキスト ファイル UTF-8形式ファイル UTF-16形式ファイル	<p>テキストファイルは、ASCIIまたはUnicode文字の非構造化シーケンスです。通常はテキストファイルの内容をNSStringオブジェクトにロードしますが、未処理の文字ストリームとしてテキストファイルを読み書きすることも可能です。</p> <p>NSStringクラスを使用したファイルからのテキストのロードについては、『<i>String Programming Guide</i>』を参照してください。</p>
構造化 データ ファイル	XMLファイル プロパティリスト ファイル 環境設定ファイル	<p>構造化データファイルは、通常は特殊文字セットを使用して配列された文字列ベースのデータから構成されます。</p> <p>XMLの解析の詳細については、『<i>Event-Driven XML Programming Guide</i>』を参照してください。</p>
アーカイ ブファイ ル	キーアーカイバオブジェクトを使用して作成されたバイナリファイル	<p>アーカイブは、アプリケーションの実行時のオブジェクトの永続バージョンの保存に使用されるファイル形式です。アーカイバオブジェクトは、オブジェクトの状態をエンコードし、ディスクに一度に書き込むことが可能なバイトストリームに変換します。アンアーカイバはこのプロセスを逆に行い、バイトストリームを使用してオブジェクトを再作成し、以前の状態に復元します。</p> <p>作成したドキュメントまたはその他のデータファイルにカスタムバイナリファイル形式を実装する場合、通常、アーカイブは使いやすい代替手段となります。</p> <p>アーカイブファイルを作成し、読み取る方法の詳細については、『<i>Archives and Serializations Programming Guide</i>』を参照してください。</p>
ファイル パッケー ジ	カスタムドキュメント形式	<p>ファイルパッケージは、任意の数のカスタムデータファイルを含むディレクトリですが、ユーザには単一のファイルとして表示されます。アプリケーションは、複数の異なるファイル、または異なるタイプのファイルを含む複雑なファイル形式の実装にファイルパッケージを使用します。たとえば、ファイル形式にバイナリデータファイルと1つまたは複数の画像、ビデオ、オーディオファイルを同時に含む場合、ファイルパッケージを使用します。ファイルパッケージの内容には、NSFileWrapperオブジェクトを使用してアクセスします。</p>

ファイル タイプ	例	解説
バンドル	アプリケーション プラグイン フレームワーク	バンドルは、コードとコードで使用するリソースを保存するための構造化された環境を提供します。ほとんどの場合、バンドルを操作することはありませんが、バンドルの内容は操作できます。ただし、必要に応じてバンドルを検索し、その情報を取得することができます。 バンドルとバンドルへのアクセス方法の詳細については、『 <i>Bundle Programming Guide</i> 』を参照してください。
コード ファイル	バイナリコードリ ソース 動的共有ライブラ リ	プラグインと共有ライブラリを処理するアプリケーションは、各項目の関連コードをロードし、その機能を活用できなければなりません。 コードリソースのメモリへのロード方法の詳細については、『 <i>Code Loading Programming Topics</i> 』を参照してください。
ファイル ラッパー	単一ファイルとし て表示されるファ イルのコレクション	アプリケーションはファイルラッパーを使用して、ペーストボードで使用可能な、またはデータレコードの一部として保存可能な連続的に書き込みできる方法でファイルを保存します。ファイルラッパーの詳細については、『 <i>ファイルラッパーのファイルコンテナとしての使用</i> 』（90 ページ）を参照してください。

標準のファイル形式では対応できない状況では、独自のカスタムファイル形式を作成できます。カスタムファイルの内容を読み書きする場合、データをバイトストリームとして読み書きし、それらのバイトをアプリケーションのファイル関連のデータ構造に適用します。バイトの読み書きの方法、および作成したファイル関連のデータ構造の管理方法は完全に制御できます。カスタムファイル形式を使用するファイルの読み書きの方法については、『*ファイルコーディネータを使用せずにファイルを読み書きする方法*』（78 ページ）を参照してください。

ファイルまたはディレクトリのパスの指定

ファイルまたはディレクトリの場所を指定する優先的な方法では、`NSURL`クラスを使用します。`NSString`クラスはパス作成に関連した多くの方法で使用されますが、ファイルとディレクトリの検索においてはURLの方が確実です。ネットワークリソースの処理も伴うアプリケーションの場合、URLの使用は、一種類のオブジェクトでローカルファイルシステムとネットワークサーバのいずれの項目も管理できることを意味します。

注意 パスをURLとして操作する場合は、NSURL以外にCFURLRef不透過型も使用できます。NSURLクラスは、CFURLRefタイプに対してtoll-free bridgeです。すなわちコード内でいずれのタイプも同様に使用できます。Core Foundationを使用したURLの作成および操作の方法については、『*CFURL Reference*』を参照してください。

ほとんどのURLは、適切なNSURLメソッドを使用し、項目までの経路が形成されるまでディレクトリ名とファイル名を結合して構築します。その方法で構築されたURLは、**パスベースURL**と呼ばれます。ディレクトリ階層を横断し、項目を検索するのに必要な名前がURLに含まれているためです（また文字列ベースのパスもディレクトリ名とファイル名を結合して構築しますが、結果はNSURLクラスで使われる形式とは多少異なる形式で保存されます）。パスベースのURL以外に、一意のIDを使用してファイルまたはディレクトリの場所を識別する、**ファイル参照URL**も作成できます。

以下のエントリはすべて、ユーザのDocumentsディレクトリのMyFile.txtと呼ばれるファイルの有効な参照です。

パスベースのURL： `file:///localhost/Users/steve/Documents/MyFile.txt`

ファイル参照URL： `file:///file/id=6571367.2773272/`

文字列ベースのパス： `/Users/steve/Documents/MyFile.txt`

ファイルシステムごとに、異なるセパレータ文字が使用されます。このような変動のため、URLオブジェクトはNSURLクラスで提供されるメソッドを使用して作成する必要があります。

URLオブジェクトをNSURLメソッドを使用して作成し、必要な場合にのみファイル参照URLに変換します。パスベースのURLは操作およびデバッグが簡単であるため、一般にNSFileManagerなどのクラスにより多用されます。ファイル参照URLの利点として、アプリケーションの実行中、パスベースのURLよりも安定していることが挙げられます。ユーザがFinderでファイルを移動する場合、そのファイルを参照するパスベースのURLはすべて移動直後に無効となり、新しいパスへの更新が要求されます。ただし、ファイルが同じディスクの別の場所に移動している場合、ファイルの一意のIDは変更されず、ファイル参照URLは有効な状態が継続します。

Important アプリケーションの実行中、ファイル参照URLは安全に使用できますが、次にアプリケーションを再起動するまで安全に保存および再利用することができません。システムがリブートされると、ファイルのIDが変更される場合があるためです。次にアプリケーションを起動するまで、ファイルの場所を永続的に保存する必要がある場合、“ブックマークを使用したファイルの検索”の説明に従ってブックマークを作成します。

ただし、文字列を使用してファイルを参照する必要性は残されています。この場合、NSURLクラスのメソッドを使用して、パスベースのURLからNSStringオブジェクトへの変換、またはその逆の変換を行います。パスをユーザに表示する場合、またはURLの代わりに文字列を受け付けるシステムルーチン呼び出す場合は、文字列ベースのパスを使用できます。NSURLオブジェクトとNSStringオブジェクト間の変換は、NSURLクラスのメソッドabsoluteStringを使用して行います。

NSURLとNSStringはファイルまたはディレクトリの場所のみが記述されるため、実際のファイルまたはディレクトリが出現する前に作成することができます。指定したファイルまたはディレクトリの実際の存在を評価するクラスはありません。実際には、ディスクで作成する前の、存在しないファイルまたはディレクトリのパスを作成しなければなりません。

URLと文字列の作成および操作に使用するメソッドの詳細については、『*NSURL Class Reference*』と『*NSString Class Reference*』を参照してください。

ファイルシステムの項目の検索

ファイルまたはディレクトリにアクセスする前に、その場所を認識する必要があります。ファイルとディレクトリを検索する方法はいくつかあります。

- ファイルを自分で検索する。
- ユーザに場所の指定を要求する。
- サンドボックスの内部および外部のアプリケーションについて、標準システムディレクトリでファイルを検索する。
- ブックマークを使用する。

iOSとOS Xのファイルシステムは、ファイルの配置場所に関する特定のガイドラインに準拠しているため、アプリケーションで作成または使用される項目のほとんどは、既知の場所に保存されます。いずれのシステムも、このような既知の場所の項目を検索するためのインターフェイスが提供され、アプリケーションはこれらのインターフェイスから項目を検索し、特定のファイルのパスを構築します。アプリケーションがユーザにファイルまたはディレクトリの場所の指定を要求するのは、“[OpenおよびSaveパネルの使用](#)”（62 ページ）で説明する限られた状況だけです。

ユーザへの項目の検索の要求

OSXでは、ユーザとファイルシステムとの相互作用は、必ず標準のOpenおよびSaveパネルを通じて行います。これらのパネルはユーザの割り込みが伴うため、以下のような限られた状況においてのみ使用します。

- ユーザのドキュメントを開く
- 新しいドキュメントの保存場所をユーザに照会する
- ユーザファイル（またはファイルのディレクトリ）を表示中のウインドウと関連付ける

OpenおよびSaveパネルは、アプリケーションで作成されたファイル、およびアプリケーションで内部的に使用されるファイルへのアクセスに使用できません。サポートファイル、キャッシュ、アプリケーションで生成されたデータファイルは、アプリケーション固有のファイルを保存する専用の標準ディレクトリのいずれかに保存します。

OpenおよびSaveパネルの表示およびカスタマイズ方法の詳細については、“[OpenおよびSaveパネルの使用](#)”（62 ページ）を参照してください。

アプリケーションバンドルの項目の検索

バンドルディレクトリの内部（または別の既知のバンドルの内部）のリソースファイルを検索する必要があるアプリケーションは、NSBundleオブジェクトを使用して検索する必要があります。バンドルは個々のファイルを特定の方法で編成するため、ファイルの場所をアプリケーションに認識させる必要がありません。NSBundleクラスのメソッドは、その編成を理解し、アプリケーションのリソースをオンデマンドで検索する場合に使用します。この方式には、バンドルにアクセスするためのコードを記述せずに、その内容を再編成できるという利点があります。またバンドルは現在のユーザの言語設定を活用して、適宜ローカライズされたリソースファイルを検索します。

以下のコードに、アプリケーションのメインバンドル内のj画像、MyImage.pngのURLオブジェクトを検索する方法を示しています。このコードは、ファイルの場所のみを判断し、ファイルを開きません。返されたURLをUIImageクラスのメソッドに渡し、使用する画像をディスクからロードします。

```
NSURL* url = [[NSBundle mainBundle] URLForResource:@"MyImage" withExtension:@"png"];
```

バンドル内の項目の検索方法を含む、バンドルの詳細については、『*Bundle Programming Guide*』を参照してください。アプリケーションのリソースのロードと使用方法の詳細については、『*Resource Programming Guide*』を参照してください。

標準ディレクトリの項目の検索

標準ディレクトリのいずれかでファイルを検索する必要がある場合、まずシステムフレームワークを使用してディレクトリを検索し、次に見つけたURLを使用してファイルのパスを構築します。

Foundationフレームワークでは、標準システムディレクトリを検索する複数のオプションが提供されます。これらの方法の使用により、アプリケーションがサンドボックス内に存在するか否かを問わず、正しいパスが取得できます。

NSFileManagerクラスのURLsForDirectory:inDomains:メソッドは、NSURLオブジェクトでパッケージングされたディレクトリの場所を返します。検索するディレクトリは、NSSearchPathDirectory定数です。これらの定数から、ユーザのホームディレクトリのURL、および標準ディレクトリのほとんどのURLが提供されます。

NSSearchPathForDirectoriesInDomainsメソッドの動作はURLsForDirectory:inDomains:メソッドに似ていますが、ディレクトリの場所を文字列ベースのパスとして返します。代わりにNSSearchPathForDirectoriesInDomainsメソッドを使用するようにしてください。

NSHomeDirectory関数は、ユーザまたはアプリケーションのいずれかのホームディレクトリのパスを返します（どのホームディレクトリが返されるかは、プラットフォームにより、またアプリケーションがサンドボックス内にあるかどうかにより異なります）。アプリケーションがサンドボックス内にある場合、ホームディレクトリはアプリケーションのサンドボックスを示し、それ以外の場合はファイルシステムのユーザのホームディレクトリを示します。ユーザのホームディレクトリのサブディレクトリのファイルを構築する場合、URLsForDirectory:inDomains:メソッドを代用することを検討してください。

先行するルーチンから受け取ったURLまたはパスベースの文字列を使用して、必要なファイルの場所を示した新しいオブジェクトを構築することができます。NSURLクラスとNSStringクラスはいずれも、パスコンポーネントの追加と削除、および全般的なパスの変更のためのパス関連のメソッドを提供します。リスト 2-1に示す例では、標準のApplication Supportディレクトリを検索し、アプリケーションのデータファイルを含むディレクトリの新しいURLを作成しています。

リスト 2-1 アプリケーションサポートディレクトリの項目のURLの作成

```
- (NSURL*)applicationDataDirectory {
    NSFileManager* sharedFM = [NSFileManager defaultManager];
    NSArray* possibleURLs = [sharedFM URLsForDirectory:NSApplicationSupportDirectory
                                                inDomains:NSUserDomainMask];

    NSURL* appSupportDir = nil;
    NSURL* appDirectory = nil;

    if ([possibleURLs count] >= 1) {
```

```
// 最初のディレクトリを使用（複数が返される場合）
appSupportDir = [possibleURLs objectAtIndex:0];
}

// 有効なアプリケーションサポートディレクトリが存在する場合、
// アプリケーションのバンドルIDを追加し、最後のディレクトリを指定する
if (appSupportDir) {
    NSString* appBundleID = [[NSBundle mainBundle] bundleIdentifier];
    appDirectory = [appSupportDir URLByAppendingPathComponent:appBundleID];
}

return appDirectory;
}
```

ブックマークを使用したファイルの検索

ファイルの場所を永続的に保存する必要がある場合、NSURLのブックマーク機能を使用します。**ブックマーク**は、ファイルの場所を説明する、NSDataオブジェクトにラップされた不透過型のデータ構造をとります。パスおよびファイル参照URLはアプリケーションの次の機能までに変更される可能性があるのに対し、通常、ブックマークはファイルが移動または改名された場合でも、ファイルのURLの再作成に使用できます。

既存のURLのブックマークを作成する場合は、NSURLの

`bookmarkDataWithOptions:includingResourceValuesForKeys:relativeToURL:error:` メソッドを使用します。NSURLBookmarkCreationSuitableForBookmarkFile オプションを指定すると、ディスクへの保存に適したNSDataオブジェクトが作成されます。リスト 2-2 に、このメソッドを使用してブックマークデータオブジェクトを作成する簡単な例の実装を示します。

リスト 2-2 URLの永続形式への変換

```
- (NSData*)bookmarkForURL:(NSURL*)url {
    NSError* theError = nil;
    NSData* bookmark = [url
        bookmarkDataWithOptions:NSURLBookmarkCreationSuitableForBookmarkFile
                        includingResourceValuesForKeys:nil
                        relativeToURL:nil
                        error:&theError];
}
```

```
if (theError || (bookmark == nil)) {  
    // すべてのエラーを処理  
    return nil;  
}  
return bookmark;  
}
```

NSURLのwriteBookmarkData:toURL:options:error:メソッドを使用して、ディスクに永続ブックマークデータを書き込む場合、ディスクで生成されるのはニックネームファイルです。ニックネームは、シンボリックリンクに似ていますが、実装方式が異なります。通常、ユーザがFinderからニックネームを作成するのは、システムの別の場所にあるファイルのリンクを作成する必要がある場合です。

ブックマークデータオブジェクトを元のURLに変換する場合、NSURLのURLByResolvingBookmarkData:options:relativeToURL:bookmarkDataIsStale:error:メソッドを使用します。リスト 2-3に、ブックマークを元のURLに変換するプロセスを示します。

リスト 2-3 永続ブックマークのURL形式への変換

```
- (NSURL*)urlForBookmark:(NSData*)bookmark {  
    BOOL bookmarkIsStale = NO;  
    NSError* theError = nil;  
    NSURL* bookmarkURL = [NSURL URLByResolvingBookmarkData:bookmark  
                                                                options:NSURLBookmarkResolutionWithoutUI  
                                                                relativeToURL:nil  
                                                                bookmarkDataIsStale:&bookmarkIsStale  
                                                                error:&theError];  
  
    if (bookmarkIsStale || (theError != nil)) {  
        // すべてのエラーを処理  
        return nil;  
    }  
    return bookmarkURL;  
}
```


Core Foundationフレームワークは、NSURLで提供されるブックマークインターフェイスと同等の、一連のCベースの関数を提供します。前記の関数の使い方については、『*CFURL Reference*』を参照してください。

ディレクトリの内容の列挙

特定のディレクトリ内のファイルの内容を確認する場合、そのディレクトリの内容を列挙します。Cocoaは、ディレクトリのファイルの逐次列挙、または同時列挙をサポートします。いずれのオプションを選択した場合でも、ディレクトリを慎重に列挙する必要があります。高価なファイルシステム内の多くのファイルへのアクセスを伴うためです。

ディレクトリのファイルの逐次列挙

ディレクトリのファイルの逐次列挙は、特定のファイルを検索し、検索後に列挙を停止する場合に推奨されます。ファイル単位の列挙は、項目を検索するメソッドを定義するNSDirectoryEnumeratorクラスを使用します。ただしNSDirectoryEnumerator自体は抽象クラスであるため、このクラスのインスタンスを直接作成しません。代わりに、NSFileManagerオブジェクトのenumeratorAtURL:includingPropertiesForKeys:options:errorHandler:またはenumeratorAtPath:メソッドを使用して、列挙に使用するクラスの具象インターフェイスを取得します。

Enumeratorオブジェクトは、列挙されたディレクトリ内に含まれるすべてのファイルとディレクトリのパスを返します。列挙は再帰的であり、デバイス境界を越えるため、返されるファイルとディレクトリ数は起動ディレクトリ内に表示される数よりも多くなる場合があります。EnumeratorオブジェクトのskipDescendantsメソッドの呼び出しにより、関連性のないディレクトリの内容をスキップすることができます。Enumeratorオブジェクトは、シンボリックリンクを解決せず、またディレクトリを示すシンボリックリンクの横断も試みません。

リスト 2-4に、enumeratorAtURL:includingPropertiesForKeys:options:errorHandler:メソッドを使用して、特定のディレクトリのユーザに表示されるサブディレクトリを、ディレクトリとファイルパッケージの種別を注釈して列挙する方法を示します。keys配列は、各項目の情報の先読みおよびキャッシュをEnumeratorオブジェクトに指示します。この情報を先読みすることにより、ディスクへのアクセスが1度に限定されるため、効率性が改善されます。列挙でファイルパッケージと非表示ファイルの内容を対象としない場合、これをオプション引数を使って指定します。エラーハンドラは、ブール値を返すブロックオブジェクトです。YESを返す場合、エラーの後、列挙が続行され、NOを返す場合、列挙が停止します。

リスト 2-4 ディレクトリの内容の列挙

```
NSURL *directoryURL = <#An NSURL object that contains a reference to a directory#>;

NSArray *keys = [NSArray arrayWithObjects:
    NSURLIsDirectoryKey, NSURLIsPackageKey, NSURLLocalizedNameKey, nil];

NSDirectoryEnumerator *enumerator = [[NSFileManager defaultManager]
    enumeratorAtURL:directoryURL
    includingPropertiesForKeys:keys

options:(NSDirectoryEnumerationSkipsPackageDescendants |
    NSDirectoryEnumerationSkipsHiddenFiles)
    errorHandler:^(NSURL *url, NSError *error) {
    // エラーを処理する。
    // エラー後に列挙を続行する場合はYESを返す
    return <#YES or NO#>;
}];

for (NSURL *url in enumerator) {

    // 明確性の理由でエラーチェックが省略される

    NSNumber *isDirectory = nil;
    [url getResourceValue:&isDirectory forKey:NSURLIsDirectoryKey error:NULL];

    if ([isDirectory boolValue]) {

        NSString *localizedName = nil;
        [url getResourceValue:&localizedName forKey:NSURLLocalizedNameKey
        error:NULL];

        NSNumber *isPackage = nil;
        [url getResourceValue:&isPackage forKey:NSURLIsPackageKey error:NULL];

        if ([isPackage boolValue]) {
            NSLog(@"Package at %@", localizedName);
        }
    }
}
```

```
    }  
    else {  
        NSLog(@"Directory at %@", localizedName);  
    }  
}  
}
```

列挙の間にファイル、すなわち親ディレクトリと現在のファイルまたはディレクトリの属性を判断する場合、およびサブディレクトリへの再帰を制御する場合、`NSDirectoryEnumerator`で宣言されるほかのメソッドを使用できます。リスト 2-5のコードは、ディレクトリの内容を列挙し、現在から24時間以内に更新されたファイルをリストします。ただし、RTFDファイルパッケージが出現すると、これらのパッケージへの再帰をスキップします。

リスト 2-5 最近修正されたファイルの検索

```
NSString *directoryPath = <#Get a path to a directory#>;  
NSDirectoryEnumerator *directoryEnumerator = [[NSFileManager defaultManager]  
enumeratorAtPath:directoryPath];  
  
NSDate *yesterday = [NSDate dateWithTimeIntervalSinceNow:(-60*60*24)];  
  
for (NSString *path in directoryEnumerator) {  
  
    if ([[path pathExtension] isEqualToString:@"rtfd"]) {  
        // このディレクトリを列挙しない  
        [directoryEnumerator skipDescendents];  
    }  
    else {  
  
        NSDictionary *attributes = [directoryEnumerator fileAttributes];  
        NSDate *lastModificationDate = [attributes  
objectForKey:NSFileModificationDate];  
  
        if ([yesterday earlierDate:lastModificationDate] == yesterday) {  
            NSLog(@"%@ was modified within the last 24 hours", path);  
        }  
    }  
}
```

```
}  
  
}
```

1回のバッチ操作によるディレクトリの内容の取得

ディレクトリ内のすべての項目の検査が必要になると認識している場合、その項目のスナップショットを取得し、適時、それらのスナップショットを反復操作します。バッチ操作によるディレクトリの内容の取得は、ディレクトリを列挙するもっとも効率的な方法ではありません。ファイルマネージャは、毎回ディレクトリの全体の内容をウォークスルーする必要があるためです。ただし、何らかの理由で全項目の検査を予定している場合は、項目を取得する方法は簡単です。

NSFileManagerを使用したディレクトリのバッチ列挙には、2つのオプションがあります。

- 浅い列挙を実行する場合は、
`contentsOfDirectoryAtURL:includingPropertiesForKeys:options:error:`または
`contentsOfDirectoryAtPath:error:`メソッドを使用します。
- 深い列挙を実行し、サブディレクトリのみを返す場合は、`subpathsOfDirectoryAtPath:error:`メソッドを使用します。

リスト 2-6に、`contentsOfDirectoryAtURL:includingPropertiesForKeys:options:error:`メソッドを使用してディレクトリの内容を列挙する例を示しています。URLを使用する利点の1つに、各項目の追加情報を効率的に取得できる点が挙げられます。この例では、ディレクトリの各項目のローカライズ名、作成日、タイプ情報を取得し、その情報を対応するNSURLオブジェクトに保存しています。メソッドが値を返す場合、引き続きarray変数で項目の反復処理を行い、項目で必要な操作を実行することができます。

リスト 2-6 ディレクトリの項目のリストを一度で取得する

```
NSURL *url = <#A URL for a directory#>;  
NSError *error = nil;  
NSArray *properties = [NSArray arrayWithObjects: NSURLLocalizedNameKey,  
                      NSURLCreationDateKey, NSURLLocalizedTypeDescriptionKey,  
                      nil];  
  
NSArray *array = [[NSFileManager defaultManager]  
                  contentsOfDirectoryAtURL:url  
                  includingPropertiesForKeys:properties  
                  options:(NSDirectoryEnumerationSkipsPackageDescendants |  
                           NSDirectoryEnumerationSkipsHiddenFiles)
```

```
                error:&error];  
  
if (array == nil) {  
    // エラーを処理する  
}
```

アプリケーション固有のファイルの保存場所の決定

Libraryディレクトリは、ユーザに代わってアプリケーションで作成および管理されるファイルの指定されたりポジトリです。アプリケーションがサンドボックス内にある場合、これらのディレクトリは別の場所にある可能性があると考えerべきです。このため、必ずNSFileManagerのメソッド、URLsForDirectory:inDomains:を使用して、このデータの保存に使用する特定のディレクトリを検索するドメインにNSUserDomainMaskを指定します。

- Application Supportディレクトリの定数NSApplicationSupportDirectoryを使用して、以下に使用中の<バンドルID>を付加します。
 - ユーザに代わってアプリケーションで作成および管理されるリソースとデータファイル。このディレクトリは、アプリケーションの状態情報、計算結果またはダウンロード後のデータ、あるいはユーザに代わって管理するユーザ作成データの保存にも使用できます。
 - 自動保存ファイル。
- Cachesディレクトリの定数NSCachesDirectoryを使用して、キャッシュされたデータファイル、またはアプリケーションで簡単に再作成できるファイルに、使用中の<bundle_ID>ディレクトリを付加します。
- NSUserDefaultsクラスを使用して、環境設定を読み書きします。このクラスは、自動的に環境設定を適切な場所に書き込みます。
- NSFileManagerメソッドのURLsForDirectory:inDomains:メソッドを使用して、一時ファイルを保存するディレクトリを取得します。一時ファイルは、進行中の一部の操作にすぐに使用し、後で破棄する予定のファイルです。一時ファイルは、その操作の終了後速やかに削除する必要があります。3日過ぎても一時ファイルを削除していない場合、使用中か否かに関わらず、システムで一時ファイルが削除される場合があります。

注意 これらの場所に保存するファイルは、アプリケーション間で共有されることはほとんどありません。このため、NSFileCoordinationインスタンスの作成と設定に使用するオーバーヘッドが不要です。

ファイルとディレクトリのアクセスに関するヒント

ファイルシステムは、システムプロセスを含むすべてのプロセスで共有されるリソースであるため、常に慎重に使用する必要があります。ソリッドステートディスク（SSD）ドライブを使用するシステムにおいても、ディスクからのデータ取得に遅延が伴うため、ファイル操作が多少遅くなる傾向があります。またファイルにアクセスする場合、安全に、またほかのプロセスに干渉しない方法でアクセスすることが重要です。

ファイル操作のパフォーマンスの遅延

ファイルシステムに関わる操作は、確実に必要とされる場合にのみ実行してください。ファイルシステムへのアクセスは、通常はほかのコンピュータレベルの作業よりも長い時間を要します。このため、この操作を行う前に、ディスクへのアクセスが実際に必要であるかどうかを確認してください。特に以下に注意してください。

- **保存が必要な価値がある場合にのみ、ディスクにデータを書き込みます。** 価値あるものの定義は、アプリケーションにより異なりますが、一般に、ユーザが明示的に提供する情報が該当します。たとえば、起動時にアプリケーションでデフォルトデータ構造が生成される場合、ユーザがそれらの構造を変更する場合を除き、構造をディスクに保存することができません。
- **必要な場合にのみ、ディスクからデータを読み取ります。** 言い換えると、ユーザインターフェイスに必要なデータをロードしますが、ファイルから一部のデータを取得する場合は、ファイル全体をロードしないでください。カスタムファイル形式の場合、ファイルマッピングを使用するか、ユーザインターフェイスに表示する必要のある、ファイルの数チャンクのみを読み取ります。残りのチャンクは、ユーザのデータとの相互作用に応じて、必要なときに読み取ります。構造化されたデータ形式の場合、データの読み取りを管理および最適化するCore Dataを使用します。

セキュアなコーディング方式の使用

ファイルに関連したセキュリティの脆弱性がプログラムに導入されないようにするために、準拠すべきいくつかの原則があります。

- 関数またはメソッドを呼び出した場合、結果コードを確認します。結果コードは、問題の発生を把握するために表示されます。このため結果コードには注意が必要です。たとえば、空と考えていたディレクトリの削除を試み、エラーコードが表示された場合、そのディレクトリは空ではないことが分かります。
- プロセスから排他的にアクセスできないディレクトリで作業する場合、ファイルが存在しないことを確認してから作成する必要があります。また、読み書きを予定しているファイルが、作成したファイルと同じファイルであることも確認する必要があります。
- 可能であれば必ず、パス名ではなくファイル記述子で動作するルーチンを使用してください。このようなルーチンにより、必ず同じファイルを処理できるようになります。
- ファイルを開くステップとは別のステップとして、意図的にファイルを作成し、既存のファイルではなく、作成したファイルを開いていると確認できるようにします。
- 呼び出す予定の関数が、シンボリックリンクに従っているかどうかを認識します。たとえば、通常のファイルとシンボリックリンクのいずれの場合でも、`lstat`関数によりファイルのステータスがわかりますが、`stat`関数はシンボリックリンクに従い、指定されたファイルがシンボリックリンクであった場合、リンク先ファイルのステータスを返します。したがって、`stat`関数を使用する場合、予定とは違うファイルにアクセスしている場合があります。
- ファイルを読み取る前に、ファイルに予定しているオーナーと権限が割り当てられていることを確認します。割り当てられていない場合、（ハングさせずに）適切に終了させてください。
- プロセスのファイルコード作成マーク（`umask`）を設定し、プロセスで作成されたファイルにアクセスを制限します。`umask`は、新しいファイルのデフォルト権限を変更するビットマスクです。アプリケーションの`umask`はリセットできません。プロセスは親プロセスから`umask`を継承します。`umask`の設定の詳細については、『`umask(2)` Mac OS X Developer Tools Manual Page』を参照してください。

その他のヒントとコーディング方式については、『*Secure Coding Guide*』の“Race Conditions and Secure File Operations”を参照してください。

パスの大文字小文字区別を想定

ファイル名を検索または比較する場合、必ず、基本ファイルシステムは大文字と小文字を区別すると仮定してください。OSXは、ファイルの区別に大文字小文字を使用する多くのファイルシステムをサポートしています。大文字小文字を区別しないファイルシステム（HFS+）の場合においても、ファイル名の比較に大文字小文字が使用される場合があります。たとえば、`NSBundle`クラスと`CFBundleAPI`は、名前付きリソースのバンドルディレクトリを検索する場合、大文字小文字を考慮します。

新しいファイルのファイル名拡張子をインクルード

ファイルはすべて、ファイル内の内容の種類を反映したファイル名拡張子を使います。ファイル名拡張子は、システムでファイルが開かれる方法を判断しやすくし、またほかのコンピュータまたはほかのプラットフォームとのファイルの交換を容易にします。たとえば、ネットワーク転送プログラムは多くの場合、ファイル名拡張子を使用してコンピュータ間でファイルを転送する最適な方法を判断します。

項目を表示する場合の表示名の使用

ファイルまたはディレクトリの名前をユーザインターフェイスで表示する必要がある場合、必ず項目の表示名を使用します。表示名の使用により、アプリケーションで表示される内容と、Finderやその他のアプリケーションで表示される内容を確実に一致させられます。たとえば、アプリケーションでファイルのパスが示される場合、表示名の使用により、現在のユーザの言語設定に従ってパスがローカライズされます。

表示名の詳細については、“[ファイルとディレクトリに代替名を割り当てられる](#)”（22 ページ）を参照してください。

バックグラウンドスレッドからの安全なファイルアクセス

一般に、ファイルのアクセスおよび操作に使用されるオブジェクトと関数は、アプリケーションの任意のスレッドから利用できます。ただし、スレッドプログラミングと同様に、バックグラウンドスレッドから安全にファイルを操作できることを確認してください。

- 複数のスレッドで同じファイル記述子を使用するのは避けてください。同じファイル記述子を使用すると、スレッド同士で状態が破壊される可能性があります。
- 必ずファイルコーディネータを使用してファイルにアクセスします。ファイルコーディネータは、プログラム（またはほかのプロセス）のほかのスレッドが同じファイルにアクセスすると、通知を発信します。これらの通知を使用して、ローカルデータ構造をクリーンアップしたり、ほかの関連作業を実行したりできます。
- ファイルのコピー、移動、リンク、削除を実行する場合は、`NSFileManager`の一意のインスタンスを作成します。ほとんどのファイルマネージャ操作はスレッドセーフですが、デリゲートを伴う操作は、特にバックグラウンドスレッドでそのような操作を実行する場合は、`NSFileManager`の一意のインスタンスを要求します。

ファイルコーディネータとファイルプレゼンタの役割

ファイルシステムは実行中のすべてのプロセスで共有されるため、2つのプロセス（または同じプロセス内の2つのスレッド）が同時に同じファイルで操作を試みると、問題が発生する場合があります。2つの異なるプロセスが同じファイルを操作する場合、片方のプロセスがクラッシュやデータ破壊などの、予期しない、深刻な問題の原因となりえる変更を行う場合があります。単一のプログラム内においても、ファイルを同時に操作する2つのスレッドがファイルを破壊し、使用を不可能にする可能性があります。このようなタイプのファイル競合を避けるために、OS X 10.7以降では、異なるプロセスまたは異なるスレッド間でファイルアクセスを安全にコーディネートするための、**ファイルコーディネータ**に対応しています。

ファイルコーディネータの仕事は、関係者が担当するファイルが別のプロセスまたはスレッドにより処理された場合、その関係者に通知することです。この場合の関係者は、アプリケーションのオブジェクトです。アプリケーションのすべてのオブジェクトは、自らを**ファイルプレゼンタ**として指定できます。すなわち直接ファイルを操作し、そのファイルに対するアクションが別のオブジェクトにより開始された場合に通知を必要とするオブジェクトです。アプリケーションのすべてのオブジェクトはファイルプレゼンタになることができ、ファイルプレゼンタは個々のファイルもファイルの全体のディレクトリも監視できます。たとえば、画像を処理するアプリケーションは、対応する画像ファイルのファイルプレゼンタとして、1つまたは複数のオブジェクトを指定します。アプリケーションの実行中に、ユーザがFinderから画像ファイルを削除した場合、その画像ファイルのファイルプレゼンタに通知し、ファイルの削除に対応する機会を与えます。

NSDocumentとUIDocumentを使用したファイルコーディネータとファイルプレゼンタの処理

iOSのUIDocumentクラスとOS XのNSDocumentクラスは、ユーザドキュメントのデータをカプセル化します。これらのクラスは自動的に以下の操作をサポートします。

- NSFileCoordinatorの実装。
- NSFilePresenterの実装。
- Autosaveの実装。

可能であれば、これらのクラスを使用してユーザデータを保存する方法を研究してください。アプリケーションはユーザのデータファイル以外のファイルを処理する場合にのみ、ファイルコーディネータとファイルプレゼンタを直接操作します。

Application Support、Cache、または一時ディレクトリのプライベートファイルを読み書きする場合は、これらのファイルは非公開と見なす必要があるため、ドキュメントクラスを使用するアプリケーションはファイルコーディネータの使用に配慮する必要はありません。

ファイルコーディネータを使用した安全な読み書き操作の保証

アプリケーションがファイルを修正する必要があると、ファイルコーディネータオブジェクトを作成し、関連するファイルプレゼンタに修正の意図を通知する必要があります。ファイルコーディネータは、アプリケーションのファイルに対する意図を、関連するすべてのファイルプレゼンタに伝達します。すべてのファイルプレゼンタが応答した後、ファイルコーディネータは、実行する必要がある実際のアクションを含むブロックを実行します。これらの処理は同時に行われるため、アプリケーションのスレッドは明示的に待機する必要はありません。

ファイルコーディネータのサポートをコードに追加するステップは、以下のとおりです。

1. ファイルまたはディレクトリを管理する各オブジェクトについて、対応するクラスに `NSFilePresenter` プロトコルの採用を指定します。監視対象のファイルに変更が行われると、システムはこのプロトコルのメソッドを使用して、オブジェクトに変更を通知します。
2. ファイルプレゼンタオブジェクトを実行時に初期化する場合、`NSFileCoordinator` の `addFilePresenter:` クラスを呼び出して、速やかにオブジェクトを登録します。
3. ファイルプレゼンタオブジェクトがファイルまたはディレクトリに対して、独自のアクションを実行する場合、`NSFileCoordinator` オブジェクトを作成し、実行する予定のアクションに対応したメソッドを呼び出します。
4. ファイルプレゼンタオブジェクトの `dealloc` メソッドで、`NSFileCoordinator` の `removeFilePresenter:` クラスメソッドを呼び出し、このオブジェクトのファイルプレゼンタの登録を解除します。

ファイルプレゼンタにより監視するファイルの選択

アプリケーションがアクセスするすべてのファイルに、監視が必要になるわけではありません。アプリケーションは常に、以下のタイプの項目を監視する場合にファイルプレゼンタを使用します。

- ユーザのドキュメント
- アプリケーションで管理されるメディアファイル（音楽、写真、ムービーなど）を含むディレクトリ
- リモートサーバ上のファイル

一般に、アプリケーションバンドルのファイル、またはアプリケーションに公開されず、~/Libraryディレクトリのアプリケーション固有のサブディレクトリに保存されるファイルは、監視の必要がありません。アプリケーションがサンドボックス内で実行される場合も、サンドボックスディレクトリ内で作成したファイルを監視する必要がありません。

ファイルプレゼンタオブジェクトの実装

アプリケーションのファイルプレゼンタオブジェクトの大半は、コントローラオブジェクトかデータモデルオブジェクトのいずれかになります。通常はコントローラオブジェクトが選択されます。コントローラオブジェクトは、一般に、データモデルオブジェクトを含むほかのオブジェクトの作成と修正を調整するためです。ただし、数100または数1000単位のファイルがアプリケーションで監視される場合、個々のファイルの管理のサポートをデータモデルオブジェクトに移行した方がよい場合があります。

ファイルプレゼンタオブジェクトを実装する場合、そのオブジェクトのクラスをNSFilePresenterプロトコルに適合させます。このプロトコルのメソッドは、ファイルまたはディレクトリで実行できる操作のタイプを示します。各種のメソッドの実装内で、オブジェクトが応答し、必要なアクションを実行します。プロトコルのすべてのメソッドを実装する必要はありませんが、オブジェクトのファイルまたはディレクトリとの相互作用に影響するメソッドはすべて実装すべきです。プロトコルのプロパティとメソッドを使用して、以下を実行します。

- 監視対象のファイルまたはディレクトリのURLを指定します（すべてのファイルプレゼンタが、presentedItemURLプロパティを実装して指定する必要があります）。
- ファイルまたはオブジェクトの制御を一時的に放棄し、別のオブジェクトによる読み書きを可能にします。
- 別のオブジェクトがファイルからの読み取りを試みる前に、保存されていない変更を保存します。
- ファイルまたはディレクトリの内容または属性の変更を追跡します。
- ファイルまたはディレクトリが移動または削除された場合、データ構造を更新します。
- ファイルプレゼンタのメソッドを実行するディスパッチキューを指定します。

ドキュメントベースのアプリケーションを実行している場合、ファイルプレゼンタのセマンティクスをNSDocumentサブクラスに組み込む必要はありません。NSDocumentクラスは、すでにNSFilePresenterプロトコルに従い、適切なメソッドを実装しています。次に、すべてのドキュメントは対応するファイルのプレゼンタとして登録され、変更の保存やドキュメントの変更の追跡などの処理を実行します。

ファイルプレゼンタのすべてのメソッドは、軽量で迅速な実行が要求されます。ファイルプレゼンタが変更に応答する場合（`presentedItemDidChange`、`presentedItemDidMoveToURL:`、`accommodatePresentedItemDeletionWithCompletionHandler:`メソッドの場合など）、ファイルプレゼンタメソッドに直接変更を組み込むことを避けた方がよいでしょう。代わりに、ブロックを非同期にディスパッチキューにディスパッチし、後で変更を処理します。これによりアプリケーションの都合のよいときに変更を処理することができ、変更を開始したファイルコーディネータの処理を不当に遅らせる必要はありません。当然ですが、ファイルの制御を保存または放棄する場合（`relinquishPresentedItemToReader:`、`relinquishPresentedItemToWriter:`、`savePresentedItemChangesWithCompletionHandler:`メソッドの場合など）、必要なすべてのアクションは後に引き延ばさずに、即時実行する必要があります。

`NSFilePresenter` プロトコルのメソッドの実装方法の詳細については、『*NSFilePresenter Protocol Reference*』を参照してください。

ファイルプレゼンタのシステムへの登録

アプリケーションで作成されたファイルプレゼンタオブジェクトはすべて、システムに登録する必要があります。オブジェクトの登録により、オブジェクトが現在特定のファイルに関連しており、ファイルへのアクションがあれば通知を必要としていることがシステムに伝えられます。ファイルプレゼンタは、`NSFileCoordinator`の`addFilePresenter:`クラスメソッドを使用して登録します。

ファイルプレゼンタとして登録するオブジェクトは、`removeFilePresenter:`メソッドを使用して適時登録を解除する必要があります。特に、コード内でファイルプレゼンタオブジェクトを解放する場合、実際に割り当てが解除される前に登録を解除する必要があります。この操作を怠ると、プログラマエラーが起こります。

ファイルコーディネータによるファイル変更の開始

ファイルプレゼンタオブジェクトが監視対象ファイルに対して変更を行う場合、事前に`NSFileCoordinator`オブジェクトを作成し、変更の種類を指定する必要があります。ファイルコーディネータは、ファイルプレゼンタにシステムレベルで変更を通知する際に使用されるメカニズムです。このオブジェクトはシステムと連動して、ほかのスレッドとプロセスに通知し、それらのスレッドとプロセスのファイルプレゼンタに応答の機会を与えます。関連するすべてのファイルプレゼンタが応答した後、実際のアクションを実行するために作成したコードのブロックをファイルコーディネータが実行します。ファイルコーディネータオブジェクトを使用するには、以下の手順を実行します。

1. `NSFileCoordinator`クラスのインスタンスを生成してください。
2. ファイルまたはディレクトリで実行するアクションに対応した、インスタンスメソッドを呼び出します。

- `coordinateReadingItemAtURL:options:error:byAccessor:` メソッドを呼び出して、ファイルまたはディレクトリの内容を読み取ります。
- `coordinateWritingItemAtURL:options:error:byAccessor:` メソッドを呼び出して、ファイルに書き込むか、ディレクトリの内容を変更します。
- `coordinateReadingItemAtURL:options:writingItemAtURL:options:error:byAccessor:` メソッドを呼び出して、ファイルまたはディレクトリの読み書きを行います。
- `itemAtURL:didMoveToURL:` メソッドを呼び出して、ファイルまたはディレクトリを新しい場所に移動します。

上記のメソッドを呼び出す場合、ファイルまたはディレクトリの実際の読み書きを実行するためのブロックを指定します。複数の項目を一括で読み取る、または書き込む必要がある場合、代わりに

`prepareForReadingItemsAtURLs:options:writingItemsAtURLs:options:error:byAccessor:` メソッドを呼び出し、そのメソッドに渡されたブロックを使用して、個々のファイルの読み書きを調整します。上記のメソッドを使用してファイルを処理する方が、ファイルを個別に読み取る、または書き込むよりもかなり効率的です。

3. 処理の後、速やかにファイルコーディネータオブジェクトを解放します。

`NSFileCoordinator` のメソッドの使い方の詳細については、『*NSFileCoordinatorClassReference*』を参照してください。

iCloudのファイル管理

iCloudストレージAPIでは、アプリケーションがユーザのドキュメントおよびデータを中央の場所に書き込み、ユーザのすべてのコンピュータおよびiOSデバイスからそれらの項目にアクセスできます。iCloudを利用して、ユーザドキュメントをどこからでもアクセスできるようにすれば、明示的に同期や転送の操作をすることなく、どのデバイスからでもドキュメントを表示し、編集できるようになります。ドキュメントをユーザのiCloudアカウントに格納すれば、安全性も確保されます。ユーザがデバイスを紛失しても、そのデバイスのドキュメントはiCloudストレージ上にあるので、損失することはありません。

Important iCloud APIは、OS Xのガベージコレクションと連携動作しません。既存のコードでガベージコレクションを利用していた場合、代わりにARCを使うよう修正する必要があります。

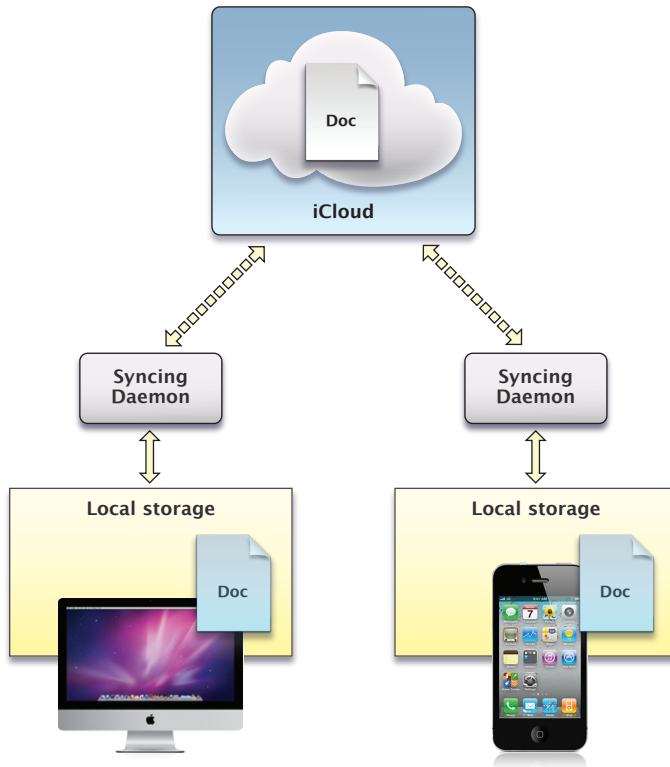
iCloudのドキュメントの保存と使い方

iCloudのドキュメントは、ユーザのコンピュータとiOSデバイスに更新を送信できる中央の送信元として機能します。すべてのドキュメントは最初にローカルディスク上で作成され、そのあとユーザのiCloudアカウントに移動されなければなりません。ただし、iCloudストレージをターゲットとするドキュメントは、すぐにiCloudに移動しません。まずドキュメントはファイルシステム内の現在の場所から、システムで管理されるローカルのディレクトリに移されます。このディレクトリはiCloudサービスで監視できます。その転送の後、可能な限り早急に、ファイルはiCloudおよびユーザのほかのデバイスに転送されます。

図4-1に示すように、iCloudストレージ内にドキュメントが存在する間に特定のデバイスで行われた変更はローカルに保存され、その後ローカルデーモンを使ってiCloudに反映されます。多くの競合する変更が同時に実行されるのを防ぐために、アプリケーションでは変更の実行にファイルコーディネータオブジェクトの使用が予定されます。ファイルコーディネータは、ドキュメントのiCloudとのやり

取りを効率化するデーモンとアプリケーションの間で変更を仲介します。このように、ファイルコーディネータはドキュメントに対してロックメカニズムと同様に振る舞い、アプリケーションとデーモンが同時にドキュメントを変更するのを防ぎます。

図 4-1 ドキュメントに施した変更をiCloudに反映

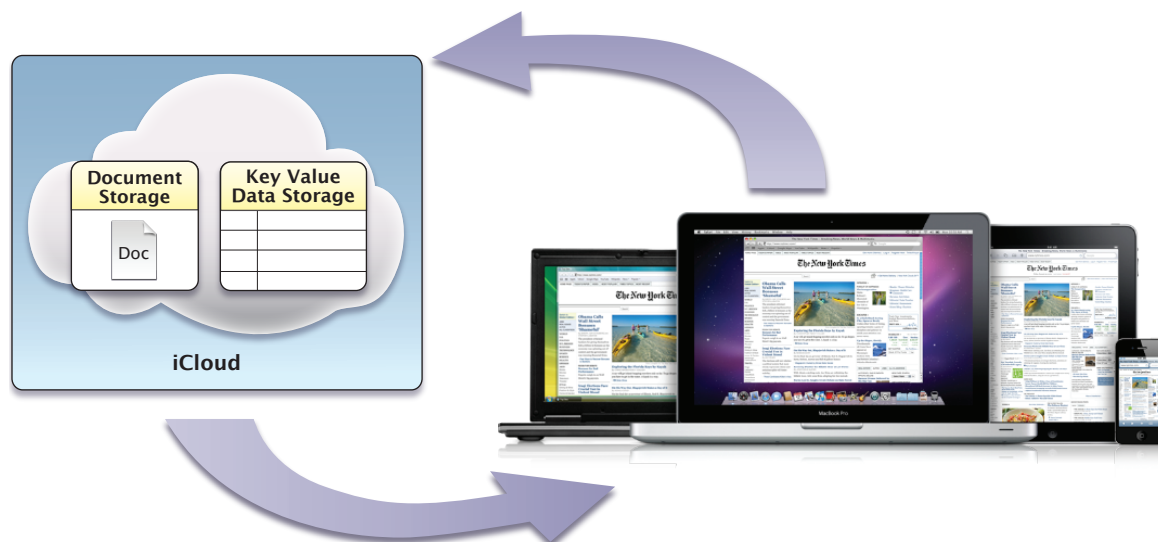


実装の面から見た場合、`NSDocument`クラスの使用がiCloudでもっとも簡単にドキュメントを管理する方法です。このクラスは、iCloudに保存されたファイルの読み書きに必要な大量の作業のほとんどを実行します。特に、`NSDocument`クラスはファイルコーディネータの作成と利用を処理し、ドキュメントを修正します。このクラスはまた、ほかのデバイスから送信されるドキュメントの変更をシームレスに統合します。さらにこのクラスは、2つのデバイスが競合的に同じファイルの更新を管理する場合に生じる、潜在的な衝突の処理をサポートします。アプリケーションのドキュメントの管理には`NSDocument`クラスの使用が要求されませんが、このクラスの使用により作業の負担が軽減します。

iCloudストレージAPI

アプリケーションの多くはiCloudドキュメントストレージを使い、ユーザのiCloudアカウントに置かれているドキュメントを共有します。iCloudストレージと聞いて普通思い浮かべる機能はこれでしょう。ユーザは、ドキュメントがデバイス間で共有されているかどうか、だけを意識していればよく、

どのデバイスからでもドキュメントを表示、管理できます。一方、iCloudキー値データストアは、ユーザの目に触れるものではありません。アプリケーションがごく少量（数十キロバイト程度）のデータを、同じアプリケーションのほかのインスタンスと共有するためのものです。アプリケーションは、状態に関する重要な情報の保存にもこの機能を使用します。雑誌閲覧アプリケーションは、ユーザが最後に閲覧した発行No.とページを保存し、株式アプリケーションはユーザが追跡する株式コードを保存します。



以下のセクションでは、iCloudストレージのさまざまな機能をアプリケーションに実装する方法を詳細に説明します。特定のクラスとインターフェイスの使い方については、対応するリファレンスドキュメントを参照してください。

iCloudのドキュメントの操作

iCloudのドキュメントをアプリケーションで読み書きする必要がある場合、協調的に実行されなければなりません。どのような瞬間にも、ローカルファイルの操作を試みるアプリケーションがほかにもあるはずです。iCloudとの間でドキュメントを転送するデーモンも、定期的にローカルファイルを操作する必要が生じます。アプリケーションがデーモンに干渉する（またはその逆）のを防ぐために、システムは、iCloudストレージのターゲットとするファイルとディレクトリを処理する、協調ロックメカニズムを提供します。

iCloudロックメカニズムの中心になるのが、ファイルコーディネータとファイルプレゼンタです。ファイルの読み書きが必要になる場合、`NSFileCoordinator`クラスのインスタンスである**ファイルコーディネータ**を使用して行います。ファイルコーディネータの仕事は、同じドキュメントでアプリケーションと同期デーモンにより実行される読み取りと書き込みを調整することです。たとえば、アプリ

ケーションとデーモンは同時にドキュメントを読み取ることができますが、ファイルへの書き込みは同時に実行できません。また、特定のプロセスがドキュメントを読み取っている場合、読み取りが終了するまで、別のプロセスによるドキュメントへの書き込みは禁止されます。

ファイルコーディネータは、調整操作以外に、ファイルプレゼンタと連動して、変更が行われる直前であることをアプリケーションに通知します。**ファイルプレゼンタ**は、`NSFilePresenter`プロトコルに従う任意のオブジェクトであり、アプリケーションで特定のファイル（またはファイルのディレクトリ）を管理する責任を担います。ファイルプレゼンタの仕事は、専用のデータ構造の完全性を保護することです。これは別のファイルコーディネータからのメッセージをリスニングし、それらのメッセージを使って内部データ構造を更新する方法で行います。ほとんどの場合、ファイルプレゼンタはアクションが要求されません。ただし、新しいURLへのファイルの移動が、直前にファイルコーディネータにより宣言された場合、ファイルプレゼンタは古いURLをファイルコーディネータから提供される新しいURLに置換する必要があります。基本ファイルまたはファイルパッケージへの変更を追跡するファイルプレゼンタの例に、`NSDocument`クラスが挙げられます。

以下に、iCloudのドキュメントの操作のために、アプリケーションで実行すべき項目のチェックリストを示します。

- iCloudの各ドキュメントをファイルプレゼンタを使用して管理します。この操作で奨励されるのは、`NSDocument`クラスを使用する方法ですが、必要に応じてカスタムファイルプレゼンタを定義することもできます。1つのファイルプレゼンタを使って、ファイルパッケージまたはファイルのディレクトリを管理できます。
- ファイルプレゼンタの作成後、`NSFileCoordinator`の`addFilePresenter:`クラスメソッドを呼び出して、ファイルプレゼンタを登録します。登録は必須です。登録されたプレゼンタオブジェクトにのみ、システムから通知が行われます。
- ファイルプレゼンタの削除の前に、`NSFileCoordinator`の`removeFilePresenter:`メソッドを呼び出して、ファイルプレゼンタの登録を解除します。

ファイル関連の操作はすべて、ファイルコーディネータオブジェクトを通じて実行されなければなりません。ドキュメントの読み書き、またはドキュメントの移動や削除には、以下の手順を使用します。

1. `NSFileCoordinator`クラスのインスタンスを作成し、ファイル操作を実行する直前のファイルプレゼンタオブジェクトを使って、インスタンスを初期化します。
2. `NSFileCoordinator`オブジェクトのメソッドを使用して、ファイルを読み書きします。
 - 1つのファイルの全部または一部を読み取る場合は、`coordinateReadingItemAtURL:options:error:byAccessor:`メソッドを使用します。
 - ファイルへの書き込みまたはファイルの削除を行う場合は、`coordinateWritingItemAtURL:options:error:byAccessor:`メソッドを呼び出します。

- 読み取りまたは書き込み操作を連続的に実行する場合は、`coordinateReadingItemAtURL:options:writingItemAtURL:options:error:byAccessor:`メソッドを使用します。
- 複数のファイルに書き込む場合、またはファイルを新しい場所に移動する場合は、`coordinateWritingItemAtURL:options:writingItemAtURL:options:error:byAccessor:`メソッドを使用します。

実際のファイル関連の操作はブロックで実行し、上記のメソッドに渡します。操作は可能な限り迅速に実行し、同時にファイルへのアクセスを試みているほかのアプリケーションをブロックしないようにしてください。

3. 操作が終了した後、ファイルコーディネータオブジェクトを解放します。

ファイルコーディネータとファイルプレゼンタの詳細については、“[ファイルコーディネータとファイルプレゼンタの役割](#)”（48 ページ）を参照してください。



Warning iCloudのファイルとディレクトリを操作する場合、できる限り英数字セットを使い、特殊な句読点やその他の特殊文字の使用は避けてください。ファイル名は大文字と小文字が区別されると仮定し、ファイルの大文字と小文字の種別を変更することはできません。必ず元のファイルと同じ文字種を使って、ファイルにアクセスしてください。このように、簡潔なファイル名にしておけば、どのようなファイルシステムでも問題なく処理できます。

ドキュメントをiCloudストレージに移動

iCloudストレージにドキュメントを移動する手順を以下に示します。

1. ファイルを作成し、適切なローカルディレクトリに保存します。
2. `NSDocument`クラスを使用してファイルを管理していない場合、ファイルの管理を担当するファイルプレゼンタを作成します。ファイルプレゼンタの詳細については、“[iCloudのドキュメントの操作](#)”を参照してください。
3. ユーザのiCloudストレージに、ファイルの移動先を指定する`NSURL`オブジェクトを作成します。
アプリケーションに対応したコンテナディレクトリのいずれかに、ファイルを配置する必要があります。`NSFileManager`の`URLForUbiquityContainerIdentifier:`メソッドを呼び出して、ディレクトリのルートURLを取得し、次にそのURLに追加ディレクトリとファイル名を付加します（アプリケーションが該当する権限を持つ任意のコンテナディレクトリに、ドキュメントを配置することができます）。
4. `NSFileManager`クラスの`setUbiquitous:itemAtURL:destinationURL:error:`メソッドを呼び出し、iCloudの指定された移動先にファイルを移動します。

iCloudにドキュメントを移動する場合、コンテナディレクトリ内部に、ファイルを管理するための追加サブディレクトリを作成できます。Documentsサブディレクトリを作成し、ユーザドキュメントの保存にそのディレクトリを使用することを強くお勧めします。iCloudでは、ドキュメントを個別に削除できるように、Documentsディレクトリの内容はユーザに表示されます。Documentsディレクトリの内部の内容はすべてグループ化され、ユーザ側での保存または削除が可能な単一のエンティティとして処理されます。ユーザのiCloudストレージのサブディレクトリは、通常のディレクトリと同様に、NSFileManagerクラスのメソッドを使用して作成します。

iCloudにドキュメントを移動した後、ドキュメントの所定の場所のURLを永続的に保存する必要はありません。NSDocumentオブジェクトを使用してドキュメントを管理する場合、そのオブジェクトにより、ローカルデータ構造は自動的にドキュメントの新しいURLで更新されます。ただし、同オブジェクトはそのURLをディスクに保存せず、カスタムファイルプレゼンタも使用できません。ユーザのiCloudストレージ内にあるドキュメントを移動できるため、ドキュメントの検索にはNSMetadataQueryオブジェクトを使用してください。検索により、ドキュメントのアクセスに適した正しいURLが保証されます。iCloudのドキュメントの検索方法の詳細については、「iCloudストレージAPI」を参照してください。

NSFileManagerクラスのメソッドの詳細については、『*NSFileManager Class Reference*』を参照してください。

iCloudのドキュメントの検索

iCloudストレージのドキュメントを検索する場合、アプリケーションは検索にNSMetadataQueryオブジェクトを使用しなければなりません。検索はユーザのiCloudストレージのドキュメントを検索する確実な方法です。URLを永続的に保存する代わりに、必ずクエリオブジェクトを使用します。アプリケーションが実行されていないときに、ユーザがiCloudストレージからファイルを削除できるためです。クエリを使用した検索は、ドキュメントの正確なリストを確保するための唯一の方法です。

NSMetadataQueryクラスはドキュメントに対して以下の検索範囲をサポートします。

- NSMetadataQueryUbiquitousDocumentsScope定数は、Documentsディレクトリ内の任意の場所にあるiCloudのドキュメントの検索に使用します（任意のコンテナディレクトリに対して、ユーザにアクセスが許可されているドキュメントはDocumentsサブディレクトリ内に配置します）。
- NSMetadataQueryUbiquitousDataScope定数は、Documentsディレクトリ以外にあるiCloudのドキュメントの検索に使用します（任意のコンテナディレクトリに対して、アプリケーションで共有を必要とする、ただしユーザの直接の操作を許可しないユーザ関連のデータファイルの保存に、この範囲を使用します）。

メタデータクエリオブジェクトを使用してドキュメントを検索する場合は、新しいNSMetadataQueryオブジェクトを作成して、以下を実行します。

1. クエリの選択範囲を、適切な値（複数の値も可）に設定します。ローカルのファイルシステムの検索とiCloudの検索との結合は不可能であることに注意してください。検索は個別に実行しなければなりません。
2. 述語を追加して、検索結果を絞ります。たとえば、すべてのファイルを検索する場合は、述語をNSMetadataItemFSNameKey == *の形式で指定します。
3. クエリ通知を登録し、並べ替え記述子や通知間隔など、その他の関連のあるクエリパラメータを設定します。

NSMetadataQueryオブジェクトは、通知を使用してクエリ結果を送信します。最低でもNSMetadataQueryDidUpdateNotification通知を登録する必要がありますが、結果収集プロセスの開始と終了を検出するためにほかの項目を登録することもできます。

4. クエリオブジェクトのstartQueryメソッドを呼び出します。
5. クエリオブジェクトが結果を生成できるように、現在の実行ループを実行します。

アプリケーションのメインスレッドでクエリを開始している場合、結果を返し、メインスレッドのイベント処理を続行します。二次スレッドでクエリを開始している場合、実行ループを明示的に設定し、実行する必要があります。実行ループの実行の詳細については、『*Threading Programming Guide*』を参照してください。

6. 通知ハンドラメソッドで結果を処理します。

結果を処理する場合、必ず最初に更新を無効にしてください。更新の無効化により、結果リストを使用している間に、クエリオブジェクトによる同リストの修正を防ぐことができます。結果の処理が終了したら、更新を再度有効にし、新しい更新の到着を許可します。

7. 検索を停止できる状態になれば、クエリオブジェクトのstopQueryメソッドを呼び出します。

メタデータクエリの作成および実行方法の詳細については、『*File Metadata Search Programming Guide*』を参照してください。

ファイルの版の食い違いの処理

アプリケーションの複数のインスタンス（異なるコンピュータまたはiOSデバイス上で実行）が、iCloudの同じドキュメントの修正を試みた場合、衝突が起こることがあります。たとえば、2つのiOSデバイスがネットワークに接続されておらず、ユーザが両デバイスに変更を行うと、いずれのデバイスもネットワークに再接続されたときに、iCloudに変更内容を反映させようと試みます。この時点では、

iCloudは同じファイルの2つの版を保有しているため、版の処理について決定する必要があります。この場合、最後に修正されたファイルを**現行のファイル**に指定し、ファイルのほかの版に**食い違いのある版**のマークを設定して対処します。

アプリケーションには、そのファイルプレゼンタオブジェクトを通じて版の食い違いが通知されます。食い違いが生じた場合、解消する最適な方法を決定する作業は、ファイルプレゼンタが担当します。アプリケーションは、可能であれば暗黙的に食い違いを解消することが要求され、そのためにファイルの内容を統合するか、古いデータが不要であれば古い版を破棄します。ただし、ファイルの内容の破棄または統合が不可能である場合、または結果的にデータが損失する可能性がある場合、適切な一連のアクションの選択の支援をアプリケーションからユーザに要求する必要があります。たとえば、保存するファイルの版の選択をユーザに委ねることもできますが、古い版を新しい名前で保存することを提案しても構いません。

アプリケーションは、できるだけ早期に食い違いを解消するよう努めなければなりません。版の食い違いが存在する場合、アプリケーションで解消されるまで、すべての版がユーザのiCloudストレージに（およびコンピュータとiOSデバイスのローカル上に）残ります。ファイルの現行の版と食い違いのある版は、`NSFileVersion`クラスのインスタンスを使用してアプリケーションに報告されます。

食い違いを解消する手順を以下に示します。

1. 現行のファイルを、`NSFileVersion`のクラスメソッド`currentVersionOfItemAtURL:`を使って取得します。
2. 次に、食い違いのある版の配列を、`NSFileVersion`のクラスメソッド`unresolvedConflictVersionsOfItemAtURL:`で取得します。
3. 食い違いのある版の各オブジェクトについて、食い違いを解決するために必要なアクションを実行します。以下のオプションから選択できます。
 - 実行可能な場合、食い違いのある版を現行のファイルに自動的に統合する。
 - 最終的にデータ損失が起こらない場合、食い違いのある版を無視する。
 - ユーザに保存する版（現行のファイルまたは食い違いのある版）を選択させる。これは最後の手段です。
4. 必要に応じて、現行のファイルを更新します。
 - 現行のファイルを最終的に選択するのであれば、更新の必要はありません。
 - 食い違いのある版を選択した場合は、協調書き込みにより、現行のファイルにその内容を上書きします。
 - 食い違いのある版を別名で保存するようユーザが指定した場合は、当該版の内容を収容する、新たなファイルを作成します。
5. 食い違いのある版に対応するオブジェクトの`resolved`プロパティをYESとします。

このプロパティをYESに設定すると、食い違いのある版のオブジェクト（および対応するファイル）は、ユーザのiCloudストレージから削除されます。

ファイルの版とその使い方の詳細については、『*NSFileVersion Class Reference*』を参照してください。

iCloudストレージの信頼性のある使い方

iCloudストレージの機能を組み込んだアプリケーションは、データを保存する際の信頼性に優れています。各ユーザのアカウントに割り当てられた容量には上限があり、これをすべてのアプリケーションが共有しています。また、ユーザは、あるアプリケーションで現在までに作成したドキュメントやデータの容量を調べ、必要に応じ、選択して削除することができます。そのため、どのようなファイルが保存されているかをきちんと管理できることが、アプリケーションにとっては重要です。ドキュメントを適切に扱うためのヒントをいくつか示します。

- すべてのドキュメントを保存するのではなく、iCloudアカウントで保存するドキュメントの選択をユーザに委ねます。ユーザが大量のドキュメントを作成する場合、iCloudにすべてのドキュメントを保存するとユーザが利用できるスペースに収まり切らなくなります。iCloudに保存するドキュメントを指定する機会をユーザに与えることにより、ユーザは利用可能なスペースを最適に利用する方法をより柔軟に決定できます。
- ドキュメントを削除すると、ユーザのiCloudアカウントからも、コンピュータやデバイスからも、一斉に削除されることに注意してください。このことをユーザに意識させ、削除操作に当たっては確認を求めるようにしてください。アプリケーションでドキュメントのローカルコピーを更新する場合、`NSFileManager`の`evictUbiquitousItemAtURL:error:`メソッドを使用します。
- ドキュメントをiCloudに保存する際は、できるだけDocumentsディレクトリに置いてください。Documents内に置いたドキュメントは、必要に応じて個別に削除し、空き容量を広げることができます。一方、このディレクトリ以外に置いたものはすべてデータとして扱われ、一斉に削除することしかできません。
- キャッシュその他、アプリケーションが内部的に用いるファイルを、ユーザのiCloudストレージに置かないでください。ユーザのiCloudアカウントは、ユーザが意識的に作成するドキュメントやデータで、アプリケーションが再作成できないものを保存するためにのみ使います。

OpenおよびSaveパネルの使用

OSXのユーザのドキュメントとファイル进行操作する場合、ユーザはファイルをファイルシステムのどこに置くかを決定する必要があります。標準のOpenパネルとSaveパネルは、ユーザのファイルと相互作用する場合に必ず使用するインターフェイスを提供します。Openパネルは、ユーザによる1つまたは複数の既存のファイルまたはディレクトリの選択が必要な場合に表示します。Saveパネルは、新しいユーザドキュメントをディスクに書き込む必要がある場合に使用します。

Important iOSアプリケーションは、ファイルの場所をユーザに問い合わせる場合にOpenまたはSaveパネルを使用しません。アプリケーションは必ずアプリケーションサンドボックスの既知の場所にファイルを保存し、カスタムインターフェイスを使用してユーザドキュメントの選択を管理します。

Openパネル：既存のファイルまたはディレクトリの取得

1つまたは複数のファイルまたはディレクトリの選択をユーザに要求する場合、`NSOpenPanel`クラスを使用します。このクラスの詳細な使用方法は、選択した項目で実行する予定のアクションにより異なります。

- 新しいウィンドウでドキュメントを開く場合は、アプリケーションに関してモーダルにOpenパネルを表示します。
- ファイルまたはディレクトリを選択し、すでに開いているドキュメントまたはウィンドウと連動して使用する場合、対応するウィンドウにアタッチされたシートとしてパネルを表示します。

表示予定のユーザドキュメントを新しいウィンドウで開く

新しいウィンドウで表示するドキュメントの選択をユーザに要求する必要がある場合、標準のOpenパネルをモーダルモードで表示します。ドキュメントベースのアプリケーションは、通常は自動的にこの動作を行い、メニューのOpenコマンドへの応答に必要なインフラストラクチャを提供します。このパネルを別の状況で表示する場合、パネルを表示する独自のメソッドを実装することができます。Openパネルを独自に作成し、表示するステップは以下のとおりです。

1. `NSOpenPanel`の`openPanel`クラスメソッドを使用して、Openパネルオブジェクトを取得します。
2. `beginWithCompletionHandler:`メソッドを使用してパネルを表示します。

3. 結果の処理には、終了ハンドラを使用します。

リスト 5-1に、標準のOpenパネルをユーザに表示するカスタムメソッドを示しています。このパネルは、利用可能なすべてのファイルタイプから単一のファイルを選択する操作をサポートする、デフォルトの設定オプションを使用します。beginWithCompletionHandler:メソッドは、デタッチされたウィンドウにパネルを表示し、即時復旧します。パネルはアプリケーションに関してモーダルに実行され、項目が選択されるとアプリケーションのメインスレッドで、指定された終了ハンドラを呼び出します。ドキュメントが正常に選択された後、ドキュメントを開き、ウィンドウを表示するコードを指定する必要があります。

リスト 5-1 Openパネルのユーザへの表示

```
- (IBAction)openExistingDocument:(id)sender {
    NSOpenPanel* panel = [NSOpenPanel openPanel];

    // このメソッドはパネルを表示し、即時復旧する
    // ユーザが項目を選択するか、パネルをキャンセルすると
    // 終了ハンドラが呼び出される
    [panel beginWithCompletionHandler:^(NSInteger result){
        if (result == NSFileHandlingPanelOKButton) {
            NSURL* theDoc = [[panel URLs] objectAtIndex:0];

            // ドキュメントを開く
        }
    }];
}
```

Important OS X 10.6以前では、Openパネルを表示する場合は、事前にこのパネルを保持し、パネルで処理が終了した後、解放する必要があります。openPanelメソッドは自動解放されたオブジェクトを返すため、通常、パネルは画面に表示された後、速やかに解放されます。パネルの保持により、途中でパネルの割り当てが解除され、消去されるのを防ぐことができます。パネルがウィンドウにアタッチされている場合は、パネルを保持する必要がなく、またOS X 10.7とARCを使用している場合もパネルを保持する必要がありません。

ドキュメントベースのアプリケーションでは、Openパネルの表示要求は、アプリケーションのデフォルトのレスポンスチェーンを構成する、アプリケーションのDocument ControllerオブジェクトのopenDocument:メソッドにより処理されます。独自のOpenパネルコードを実装する代わりに、

`openDocument`：メソッドの呼び出しを検討しても構いません。このメソッドにより、カスタムコードとデフォルトのアプリケーションインフラストラクチャにより、常に同じOpenパネルが表示されます。また変更は一か所でのみ実行するため、後でパネルをカスタマイズするのが容易になります。

ドキュメントベースのアプリケーションの実装の詳細については、『*Mac App Programming Guide*』を参照してください。

すでに開いているウィンドウのファイルとディレクトリの選択

現在のウィンドウに関連したファイルまたはディレクトリを選択する場合、標準のOpenパネルをChooseパネルとして設定し、ユーザの選択の取得に使用します。標準のOpenパネルとChooseパネルの主な違いは表示様式です。標準のOpenパネルはスタンドアロンモーダルパネルとして表示するのに対し、Chooseパネルはほとんどの場合、既存のウィンドウのいずれかにアタッチします。パネルのウィンドウへのアタッチにより、ウィンドウに対してモーダルになりますが、アプリケーション全体に対してはモーダルではありません。ほかにもパネル自体の設定に関連した違いがあります。Chooseパネルでは、ほかのオプションからディレクトリ、複数項目、非表示項目の選択を許可する設定が可能です。

`openPanel`メソッドで返される`NSOpenPanel`オブジェクトは、以下のデフォルトオプションで設定されます。

- ファイル選択：有効
- ディレクトリ選択：無効
- ニックネーム解決：有効
- 複数選択：無効
- 非表示ファイルの表示：無効
- ファイルパッケージのディレクトリとしての処理：無効
- ディレクトリの作成：無効

リスト 5-2に、`NSDocument`サブクラスのいずれかに実装し、一部のファイルをインポートし、そのファイルをドキュメントに関連付けるメソッドを示しています。パネルを設定した後、このメソッドは`beginSheetModalForWindow:completionHandler:`メソッドを使用して、ドキュメントのメインウィンドウにアタッチされたシートとしてパネルを表示します。ユーザが何らかのファイルを選択すると、`URLs`メソッドは組み込むための対応するファイルとディレクトリの参照を格納します。パネルがドキュメントウィンドウにアタッチされているため、コード内で明示的に保持する必要はありません。

リスト 5-2 Openパネルのウインドウとの関連付け

```
- (IBAction)importFilesAndDirectories:(id)sender {
    // ドキュメントのメインウインドウを取得する
    NSWindow* window = [[[self windowControllers] objectAtIndex:0] window];

    // パネルを作成し、設定する
    NSOpenPanel* panel = [NSOpenPanel openPanel];
    [panel setCanChooseDirectories:YES];
    [panel setAllowsMultipleSelection:YES];
    [panel setMessage:@"Import one or more files or directories."];

    // ドキュメントのウインドウにアタッチされたパネルを表示する
    [panel beginSheetModalForWindow:window completionHandler:^(NSInteger result){
        if (result == NSFileHandlingPanelOKButton) {
            NSArray* urls = [panel URLs];

            // URLに基づき、インポートする項目のリストを構築する
        }

    }]];
}
```

シングルウインドウアプリケーションの場合、**Open**パネルはアプリケーションのメインウインドウに関連付けます。ドキュメントベースのアプリケーションでは、現在のドキュメントのメインウインドウに関連付けます。

Saveパネル：新しいファイル名の取得

ドキュメントベースのアプリケーションの場合、現在のドキュメントの保存の要求は、ドキュメントオブジェクトが処理する必要があります。NSDocumentクラスは、**Save**パネルと自動保存などの保存関連の操作の両方を管理する、高度なインフラストラクチャを実装します。その結果、ドキュメントオブジェクトはデフォルトの**Save**パネルを大幅にカスタマイズすることが多くなり、既存のNSDocumentメソッドを使う場合を除き、追加的にカスタマイズを行うことは推奨されません。この場合も、別のファイル形式を使用するファイルのエクスポートの処理には、カスタム**Save**パネルを使用できます。

NSDocumentインフラストラクチャを使用しない場合、適時、独自のSaveパネルを作成し、表示することができます。NSSavePanelクラスは、デフォルトのSaveパネルを作成およびカスタマイズするためのメソッドを提供します。独自のドキュメントインフラストラクチャを実装する場合、Saveパネルの使用は、ユーザが作成し保存を必要とするドキュメントに制限してください。アプリケーションで暗黙的に作成および管理されるファイルには、Saveパネルを使用しないでください。たとえば、iPhotoはメインライブラリファイルの場所の指定をユーザに要求しません。iPhotoはユーザとの相互作用を行わずに、一般的な場所にファイルを作成します。ただし、ユーザがディスクに画像をエクスポートする場合、iPhotoはSaveパネルを表示します。

リスト 5-3に、ドキュメントオブジェクトが新しいタイプにファイルをエクスポートする場合に呼び出すメソッドを示します。このメソッドは、ドキュメントの既存の名称と保存される新しいタイプから、新しいファイル名を組み立てます。次に新しい名前がデフォルト値に設定されたSaveパネルを表示し、適宜、保存操作を開始します。

リスト 5-3 新しいタイプによるファイルの保存

```
double check that there is no NSURL method for specifying the string.
Regardless build the path using NSURL and then convert
- (void)exportDocument:(NSString*)name toType:(NSString*)typeUTI
{
    NSWindow* window = [[[self windowControllers] objectAtIndex:0] window];

    // 現在の名前と、指定されたUTIに対応するファイル拡張子を使って
    // ファイルの新しい名前を構築する
    CFStringRef newExtension = UTTypeCopyPreferredTagWithClass((CFStringRef)typeUTI,
                                                                kUTTagClassFilenameExtension);
    NSString* newName = [[name stringByDeletingPathExtension]
                        stringByAppendingPathExtension:(NSString*)newExtension];
    CFRelease(newExtension);

    // ファイルのデフォルト名を設定し、パネルを表示する
    NSSavePanel* panel = [NSSavePanel savePanel];
    [panel setNameFieldStringValue:newName];
    [panel beginSheetModalForWindow:window completionHandler:^(NSInteger result){
        if (result == NSFileHandlingPanelOKButton)
        {
            NSURL* theFile = [panel URL];
        }
    }];
}
```

```
        // 内容を新しい形式で書き込む
    }
    }];
}
```

ドキュメントベースのアプリケーションの実装の詳細については、『*Mac App Programming Guide*』を参照してください。

ユーザが選択できるファイルタイプのフィルタによる制限

アプリケーションで開くことができるのが特定のファイルタイプに限定される場合、フィルタリングにより、アプリケーションが実際にサポートするファイルのサブセットにユーザの選択を制限できます。Openパネルは、デフォルトで、ユーザが選択できるファイルのタイプを制限しません。アプリケーションで処理できないファイルの選択をユーザに禁止する場合、Openパネルに1つまたは複数のフィルタをインストールします。指定したフィルタに一致しないファイルは、Openパネルでグレイ表示され、ユーザは選択できません。また未知のファイルタイプのグレイ表示するフィルタを、Saveパネルにインストールすることもできます。

フィルタとして、UTI、ファイル名拡張子、または両方の組み合わせを指定します。フィルタ文字列を配列にまとめた後、`setAllowedFileTypes:`メソッドを使用してパネルに割り当てます。パネルはファイルの実際のフィルタリングを自動的に処理します。パネルが表示されている間に、フィルタを変更できます。

リスト 5-4に、Openパネルの設定により画像タイプの選択のみを許可する例を示します。NSImageクラスは、サポートされる画像タイプを取得する利便性の高いメソッドを提供します。返される配列の文字列はUTIであるため、結果は直接パネルに渡されます。

リスト 5-4 画像ファイルタイプのフィルタリング

```
- (IBAction)askUserForImage:(id)sender {
    NSOpenPanel* panel = [[NSOpenPanel openPanel] retain];

    // NSImageクラスでサポートされる画像の選択を
    // ユーザに委ねる
    NSArray* imageTypes = [NSImage imageTypes];
    [panel setAllowedFileTypes:imageTypes];

    [panel beginWithCompletionHandler:^(NSInteger result){
```

```
if (result == NSFileHandlingPanelOKButton) {  
    // 画像を開く  
}  
  
[panel release];  
}];  
}
```

OpenおよびSaveパネルへのアクセサリビューの追加

ファイルを開くまたは保存する際の追加オプションをユーザに提供する場合、標準のOpenおよびSaveパネルにアクセサリビューを追加します。通常、OpenおよびSaveパネルはイベントを処理し、ユーザが結果をキャンセルまたは受け入れるまで、すべての相互作用を処理します。**アクセサリビュー**の追加により、情報収集のためのカスタムコントロールとビューを追加することができます。またパネルの動作を修正することができます。たとえば、カスタムコントロールを使用すると、パネルで 사용되는フィルタセットを変更すること、あるいはファイルを開く、または保存するときにコードで 사용되는オプションを変更することができます。

OpenまたはSaveパネルにアクセサリビューを追加する基本的なプロセスを、以下に示します。

1. nibファイルからアクセサリビューをロードします（アクセサリビューはプログラムでも作成できますが、nibファイルを使用する方法が簡単です）。
2. パネルを作成します。
3. `setAccessoryView:`メソッドを使用して、ビューをパネルに関連付けます。
4. パネルを表示します。

上記のステップを通常のパネルを開くステップと比較した場合、ビューをロードし、`setAccessoryView:`メソッドを呼び出す箇所のみ異なります。アクセサリビューの管理、およびイベントへの応答に必要なその他の作業は、各自の責任において行ってください。

アクセサリビューの内容の定義

アクセサリビューは単なるビューに過ぎないため、アプリケーションのほかのビューを定義する場合と同様に定義します。アクセサリビューは、常に、任意の数のサブビューを含む単一のメインビューです。アクセサリビューは1つまたは複数のコントロールのホストとして設定されるのが、もっとも

一般的です。アクセサリビューでユーザがコントロールと相互作用する場合、そのコントロールはターゲット／アクション設計パターンを使用して、アプリケーションのカスタムコードを呼び出します。次にそのコードを使用して、設定オプションを保存するか、関連するアクションを実行します。

nibファイルは、アクセサリビューを定義するもっとも簡単な方法です。nibファイルを設定する前に、アプリケーションのどのオブジェクトをパネルに表示するかを認識する必要があります。たとえば、ドキュメントベースのアプリケーションでは、`NSDocument` サブクラスは通常はOpenおよびSaveパネルの表示に責任を持ちます。一般に、パネルを表示するオブジェクトは、アクセサリビューの内容も保有します。上記を念頭に置いた場合の、nibファイルの設定は以下のようになります。

1. シングルビューを内容とする新しいnibファイルの作成を実行します。
2. nibファイルのFile's Ownerを、パネルを表示するオブジェクトに設定します。
3. nibファイルのトップレベルのビューオブジェクトの内容を設定します。
 - チェックボックスなど、アクセサリビューに含めるサブビューの追加を実行します。
 - 必要に応じて、トップレベルのビューオブジェクトのクラスを変更できますが、これは必須ではありません。汎用ビューをメインビューとして使用する場合、基本パネルに残りの内容に適切な背景の外観が表示されます。
4. (適宜) ビューへのアクションの接続を行い、カスタムコードとの相互作用を効率化します。File's Ownerとして割り当てたオブジェクトに、アクションのメソッドを自分で実装する必要があります。
5. ユーザからデータをパッシブに収集する場合に使用するコントロールに対して、(適宜) アウトレットの接続を実行します。パネルの終了ハンドラは、アウトレットを使用してコントロールのデータにアクセスします。
6. すべてのサブビューを完全に表示している間は、アクセサリビューのサイズをできるだけ小さく設定してください。
7. nibファイルを保存します。

アクセサリビューを表示する場合、OpenおよびSaveパネルはアクセサリビュー全体を表示します。アクセサリビューがパネルよりも大きい場合、ビューに合わせてパネルが拡大します。パネルが過剰に拡大すると、小型の画面では奇妙な外観になる場合、あるいは問題が発生する場合があります。ほとんどのアクセサリビューは、コントロールの数を限定する必要があります。このため、10を超えるコントロールを追加している場合、アクセサリビューが妥当であるかどうか、また情報収集に別の方法がないかどうかを検討した方がよいでしょう。

ビューと残りのnibファイルの設定方法については、『*Xcode 4 User Guide*』を参照してください。

実行時のアクセサリビューのロードと設定

OpenまたはSaveパネルを表示する直前に、アクセサリビューをロード（または作成）し、`setAccessoryView:`メソッドを使用してパネルにアクセサリビューをアタッチする必要があります。アクセサリビューをプログラムで作成する場合、通常はパネルを表示する直前に作成します。アクセサリビューをnibファイルに保存している場合、まずそのnibファイルをロードする必要があります。

アクセサリビューのnibファイルのロードは、nibファイル自体が正しく設定されていれば比較的簡単です。nibファイルを設定するもっとも簡単な方法は、パネルをnibファイルのFile's Ownerとして表示するオブジェクトの割り当てです。その場合、オブジェクトで定義されるアウトレットとアクションは自動的に接続され、nibファイルがメモリにロードされた直後に使用可能になります。リスト 5-5に、このプロセスの例を示しています。この例のメソッドは、`NSDocument`オブジェクトで実装されています。メソッドが呼び出されると、ドキュメントのメインウィンドウにアクセサリビューをアタッチしたOpenパネルを表示します。アクセサリビュー自体には1つのチェックボックスのみ含まれ、これはドキュメントオブジェクトで定義されたカスタム`optionCheckbox` アウトレットに接続されています。（アウトレット自体はプロパティを使って実装されます。）ハンドラブロックは、チェックボックスの値に基づいて、ファイルを開く方法を判断します。

リスト 5-5 アクセサリビューを含むnibファイルのロード

```
- (IBAction)openFileWithOptions:(id)sender {
    NSOpenPanel*    panel = [NSOpenPanel openPanel];
    NSWindow*       window = [[[self windowControllers] objectAtIndex:0] window];

    // アクセサリビューを含むnibファイルをロードし、パネルにアタッチする
    if ([NSBundle loadNibNamed:@"MyAccessoryView" owner:self])
        [panel setAccessoryView:self.accessoryView];

    // Openパネルを表示し、結果を処理する
    [panel beginSheetModalForWindow:window completionHandler:^(NSInteger result){
        if (result == NSFileHandlingPanelOKButton) {
            BOOL option1 = ([self.optionCheckbox state] == NSOnState);

            // 指定されたオプションを使って、選択されたファイルを開く
        }
    }];
}
```

アクセサリビューをプログラムで作成している場合、`loadNibNamed:owner:`メソッドの呼び出しを含むIFステートメントを、カスタムビューの作成コードと置換できます。

nibファイルからオブジェクトをロードする方法の詳細については、『*Resource Programming Guide*』を参照してください。

ファイルとディレクトリの管理

ファイルとディレクトリを伴うもっとも基本的な操作に、ファイルシステムに関連した作成と移動があります。これらの操作は、タスクの実行に必要なファイルシステム構造がアプリケーションで構築されるプロセスです。ほとんどの操作は、`NSFileManager`クラスがファイルの作成および操作に必要な機能を提供します。同クラスから必要な機能が提供されない場合、これはまれですが、BSDレベルの関数を直接使用する必要があります。

新しいファイルとディレクトリのプログラムによる作成

このファイルおよびディレクトリの作成は、ファイル管理の基本的な部分です。通常、コードで生成されるファイルを編成するカスタムディレクトリを作成します。たとえば、複数のカスタムディレクトリをアプリケーションのApplication Supportディレクトリに作成し、アプリケーションで管理されるプライベートデータファイルを保存できます。またファイルを作成する方法は数多くあります。

ディレクトリの作成

カスタムディレクトリを作成する場合、`NSFileManager`のメソッドを使用して作成します。プロセスは、ディレクトリの作成の権限を持つ任意の場所に、ディレクトリを作成することができます。ここには、常に現在のホームディレクトリが含まれ、ほかのファイルシステムの場所を含めることもできます。作成するディレクトリは、ディレクトリのパスを構築し、以下のメソッドのいずれかに`NSURL`または`NSString`オブジェクトを渡して指定します。

- `createDirectoryAtURL:withIntermediateDirectories:attributes:error:` (OS X 10.7以降のみ)
- `createDirectoryAtPath:withIntermediateDirectories:attributes:error:`

リスト 6-1に、~/Library/Application Supportディレクトリ内にアプリケーションファイルのカスタムディレクトリを作成する方法を示しています。このメソッドでは、存在していないディレクトリが作成され、ディレクトリのパスが呼び出し元のコードに返されます。このメソッドは毎回ファイルシステムにアクセスするため、このメソッドを繰り返し呼び出し、URLを取得する作業を省略できます。この作業に代わり、メソッドを1度呼び出して、そのあと返されたURLをキャッシュした方がよいでしょう。

リスト 6-1 アプリケーションファイルのカスタムディレクトリの作成

```
- (NSURL*)applicationDirectory
{
    NSString* bundleID = [[NSBundle mainBundle] bundleIdentifier];
    NSFileManager* fm = [NSFileManager defaultManager];
    NSURL* dirPath = nil;

    // ホームディレクトリでアプリケーションサポートディレクトリを見つける
    NSArray* appSupportDir = [fm URLsForDirectory:NSApplicationSupportDirectory
                                         inDomains:NSUserDomainMask];

    if ([appSupportDir count] > 0)
    {
        // Application SupportディレクトリのURLに
        // バンドルIDを付加する
        dirPath = [[appSupportDir objectAtIndex:0]
URLByAppendingPathComponent:bundleID];

        // ディレクトリが存在しない場合、このメソッドにより作成される
        // このメソッド呼び出しはOS X 10.7以降でのみ機能する
        NSError* theError = nil;
        if (![fm createDirectoryAtURL:dirPath withIntermediateDirectories:YES
                                attributes:nil error:&theError])
        {
            // エラーを処理する。

            return nil;
        }
    }

    return dirPath;
}
```

OS X 10.6以前でコードを実行する必要がある場合、`createDirectoryAtURL:withIntermediateDirectories:attributes:error:`メソッドの呼び出しを、類似する`createDirectoryAtPath:withIntermediateDirectories:attributes:error:`メソッド

ドの呼び出しと置換できます。唯一異なるのは、最初のパラメータとして、URLの代わりに文字列ベースのパスを渡すことです。ただし、NSURLクラスは、このパスの文字列ベースの版を返す `path` メソッドを定義します。

NSFileManagerメソッドは、その簡潔性を理由に、新しいディレクトリを作成する方法として多用されます。ただし、ディレクトリを自分で作成する場合は、`mkdir`関数も使用できます。この関数を使用する場合、中間ディレクトリの作成、および発生するエラーの処理に責任を持つ必要があります。

新しいファイルの作成

ファイルの作成は、ファイルシステムのファイルのレコードの作成と、ファイルへの内容の埋め込みの2つのパートから構成されます。ファイル作成用の高レベルのインターフェイスはすべて、2つのタスクを同時に実行し、通常はNSDataまたはNSStringオブジェクトの内容をファイルに埋め込み、ファイルを閉じます。また下位レベルの関数を使用して、空ファイルを作成し、ファイルにデータを埋めるのに使用するファイル記述子を取得することもできます。ファイルの作成に使用できる一部のルーチンを以下に示します。

- `createFileAtPath:contents:attributes:` (NSFileManager)
- `writeToURL:atomically:` (NSData)
- `writeToURL:atomically:` (NSString)
- `writeToURL:atomically:encoding:error:` (NSString)
- `writeToURL:atomically:` (プロパティリストの書き出しのためのこのメソッドは、さまざまなコレクションクラスにより定義される)
- `open`関数と`O_CREAT`オプションにより、新しい空ファイルが作成される。

注意 作成したファイルはすべて、現在のユーザとプロセスに関連した権限を継承します。

新しいファイルの内容をすべて一度に書き込む場合、通常、システムルーチンはファイルを閉じてから内容をディスクに書き込みます。ルーチンがファイル記述子を返す場合、その記述子を使用して、ファイルの読み書きを続けることができます。ファイルの内容を読み書きする方法については、“[ファイルコーディネータを使用せずにファイルを読み書きする方法](#)” (78 ページ) を参照してください。

ファイルとディレクトリのコピーと移動

ファイルシステムに関する項目をコピーする場合は、`NSFileManager`クラスが提供する `copyItemAtURL:toURL:error:メソッド`と`copyItemAtPath:toPath:error:メソッド`を使用します。ファイルを移動する場合は、`moveItemAtURL:toURL:error:メソッド`または`moveItemAtPath:toPath:error:メソッド`を使用します。

単一のファイルまたはディレクトリを1つずつ移動またはコピーする場合は、前者のメソッドを使用します。ディレクトリを移動またはコピーする場合は、ディレクトリとその内容すべてに影響します。移動とコピー操作のセマンティクスは、**Finder**と同じです。同じボリュームで移動操作を実行しても、その項目の新しい版が作成されることはありません。ボリューム間の移動操作は、コピー操作と同様に振る舞います。項目を移動またはコピーする場合、現在のプロセスは項目を読み取り、新しい場所に移動またはコピーする権限を持つ必要があります。

移動およびコピー操作は、終了までに長時間を要する場合があります、`NSFileManager`クラスはこれらの操作を同時に実行します。したがって、このような操作は、アプリケーションのメインスレッドではなく、並列ディスパッチキューで実行することが推奨されます。リスト 6-2に、仮想アプリケーションのプライベートデータのバックアップを非同期に作成する方法で、上記の操作を行う例を示します（この例では、プライベートデータは`~/Library/Application Support/<bundleID>/Data`ディレクトリにあります。`<bundleID>`は、アプリケーションの実際のバンドル識別子です）。ディレクトリのコピーの最初の試みに失敗すると、このメソッドは前回のバックアップが存在するかどうかを確認し、存在する場合は削除します。この後、試行を繰り返し、2回目に失敗すると中止します。

リスト 6-2 ディレクトリの非同期コピー

```
- (void)backupMyApplicationData {
    // アプリケーションのメインデータディレクトリを取得する
    NSArray* theDirs = [[NSFileManager defaultManager]
        URLsForDirectory:NSApplicationSupportDirectory
                        inDomains:NSUserDomainMask];

    if ([theDirs count] > 0)
    {
        // ~/Library/Application Support/&lt;bundle_ID&gt;/Dataのパスを構築する
        // <bundleID>は、アプリケーションの実際のバンドルID
        NSURL* appSupportDir = (NSURL*)[theDirs objectAtIndex:0];
        NSString* appBundleID = [[NSBundle mainBundle] bundleIdentifier];
        NSURL* appDataDir = [[appSupportDir URLByAppendingPathComponent:appBundleID]
            URLByAppendingPathComponent:@"Data"];
    }
}
```



```
        // ~/Library/Application Support/&lt;bundle_ID&gt;/Data.backupにデータをコピー
        する
        NSURL* backupDir = [appDataDir URLByAppendingPathExtension:@"backup"];

        // 非同期にコピーを実行する
        dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
        0), ^{
            // 後でデリゲートを追加する予定の場合に限り、
            // 移動/コピー操作にはファイルマネージャの割り当て/起動を行うようにしてください
            NSFileManager* theFM = [[NSFileManager alloc] init];
            NSError* anError;

            // ディレクトリのコピーを試みる
            if (![theFM copyItemAtURL:appDataDir toURL:backupDir error:&anError]) {
                // エラーが発生した場合、おそらく前回のバックアップディレクトリが
                // すでに存在していることが原因。古いディレクトリを削除し、再度試みる
                if ([theFM removeItemAtURL:backupDir error:&anError]) {
                    // 操作が再度失敗する場合、実際に中止する
                    if (![theFM copyItemAtURL:appDataDir toURL:backupDir error:&anError])
                    {
                        // エラーを報告する
                    }
                }
            }

        });
    }
}
```

NSFileManagerメソッドの使い方については、『*NSFileManager Class Reference*』を参照してください。

ファイルとディレクトリの削除

ファイルとディレクトリを削除する場合は、NSFileManagerクラスの以下のメソッドを使用します。

- `removeItemAtURL:error:`
- `removeItemAtPath:error:`

上記のメソッドを使用してファイルを削除する場合、ファイルをファイルシステムから永続的に削除していることに注意してください。上記のメソッドは、後で復元できるゴミ箱にファイルを移動するものではありません。

`NSFileManager`メソッドの使い方については、『*NSFileManager Class Reference*』を参照してください。

非表示ファイルの作成と処理

OS Xでは、ファイルをユーザに非表示にする場合は、ファイル名の先頭にピリオド文字を追加します。これらの文字の追加により、Finderやその他の表示関連のインターフェイスに対して、通常の条件下でファイルを非表示として処理するように指示されます。この機能の使用は、非常に限られた場合に限定してください。たとえば、カスタムディレクトリ名をローカライズするプロセスには、ディレクトリのトップレベルで、非表示ファイルをローカライズ名に置換する処理が含まれます。この技法は、ユーザのドキュメントや、アプリケーションで作成されるその他のデータの非表示に使用しないでください。

ファイルに非表示をマークしている場合でも、ユーザに表示する方法は残っています。ユーザはターミナルアプリケーションを使用して、ファイルシステムの実際の内容をいつでも表示できます。Finderと異なり、ターミナルのコマンドラインはファイルシステムの項目の表示名やローカライズ名ではなく、実際の名前を表示します。また非表示ファイルを選択して、表示または編集する作業をユーザに許可する場合は、**Open**および**Save**パネルに非表示ファイルを表示することもできます（`setShowsHiddenFiles:`メソッドを使用）。

ファイル名の先頭にピリオドが付加されているか否かを問わず、アプリケーションのコードは必ず非表示ファイルを認識します。ディレクトリの内容を列挙するメソッドは、一般に、列挙に非表示ファイルを含めます。この規則の唯一の例外は、現在のディレクトリと親ディレクトリを表すメタディレクトリ`.`と`..`です。したがって、ディレクトリの内容を列挙するコードは、通常は無視という単純な方法により、非表示ファイルの処理に対応することが要求されます。

ファイルコーデックデータを使用せずにファイルを読み書きする方法

ファイルの読み書きには、コードと基本ディスクの間でバイトシーケンスを転送する処理が伴います。これはファイル管理のもっとも低レベルの形態ですが、より高度な技法の基礎でもあります。ある意味で、もっとも高度なデータ構造も、バイトシーケンスに変換しなければディスクに保存できません。同様に、同じデータをバイトシーケンスとしてディスクから読み取らなければ、外観的により高度なデータ構造を再構築するのに使用できません。

ファイルの内容の読み書きには、複数の異なる技法があり、そのほぼすべてがiOSとOS Xのいずれでもサポートされています。本質的にはすべての技法が同じ内容ですが、多少異なる点があります。ファイルデータの連続的な読み書きを要求する技法や、ファイルの一部のみのジャンプ移動および操作を許可する技法、また、非同期の読み書きを自動的にサポートする技法や、実行の制御を強化するために同期的に読み書きを実行する技法があります。

利用可能な技法からどの方法を選択するかは、読み書きプロセスに対してどの程度制御するか、またファイル管理コードの作成にどの程度の労力を費やすかの決定の問題になります。Cocoaストリームなどの高レベルの技法では、柔軟性が制限されますが、使いやすいインターフェイスが提供されます。POSIXやGrand Central Dispatch (GCD) などの低レベルの技法は、最大限の柔軟性と性能が得られますが、コードの追記が要求されます。

ファイルの非同期の読み書き

ファイル操作にはディスク（おそらくはネットワークサーバ上のディスク）へのアクセスが伴うため、ほとんどの場合、それらの操作を非同期で実行することが推奨されます。CocoaストリームやGrand Central Dispatch (GCD) などの技法は、常に非同期の実行が構築されています。ほかの技法は非同期に用いることができないという意味ではなく、これらの技法では自動的に非同期動作が行われるという意味です。ここでは、コードの実施箇所に配慮せずに、ファイルデータの読み書きに集中できます。

ストリームを使用したファイル全体の線形処理

ファイルの内容を常に最初から最後まで読み書きする場合、ストリームのシンプルなインターフェイスが、この操作を非同期に行います。通常、ストリームはソケットや、時間の経過によりデータが利用可能になるほかの種類ソースに使用されます。ただし、ファイル全体の読み書きを1つまたは複数のバーストで実行する場合にもストリームを使用できます。ストリームは2つのタイプが利用できます。

- データを連続的にディスクに書き込む場合は、`NSOutputStream`を使用します。
- データを連続的にディスクから読み出す場合は、`NSInputStream`オブジェクトを使用します。

ストリームオブジェクトは、現在のスレッドの実行ループを使用して、ストリームの読み書き操作をスケジューリングします。読み取りを待機しているデータが存在すれば、入力ストリームは実行ループを解除し、デリゲートを通知します。データを書き込む空きスペースがある場合、出力ストリームは実行ループを解除し、デリゲートを通知します。ファイルを操作する場合、この動作が通常意味するのは、実行ループが短時間に連続的に複数回解除され、デリゲートコードからファイルデータの読み書きが可能になることです。またストリームオブジェクトを閉じるまで、または入力ストリームの場合は、ストリームがライフサイクルの終了に達するまで、デリゲートコードが繰り返し呼び出されることも意味します。

ストリームオブジェクトを設定および使用して、データを読み書きする方法の詳細および例については、『*Stream Programming Guide for Cocoa*』を参照してください。

GCDを使用したファイルの処理

Grand Central Dispatchは、ファイルの内容を非同期に読み書きする複数の異なる方法を提供します。

- ディスパッチI/Oチャンネルを作成し、このチャンネルを使ってデータを読み書きします（OS X 10.7以降。iOSでは非サポート）。
- 単一の非同期操作を実行する場合は、`dispatch_read(3)` Mac OS X Developer Tools Manual Pageまたは`dispatch_write(3)` Mac OS X Developer Tools Manual Page簡易関数を使用します（OS X 10.7以降。iOSでは非サポート）。
- ディスパッチソースを作成して、カスタムイベントハンドラブロックの実行をスケジュールします。このブロックでは、標準のPOSIX呼び出しを使用して、ファイルのデータの読み書きを行います。

ディスパッチI/Oチャンネルは、推奨されるファイルの読み書き方法です。ファイル操作の実行のタイミングを直接制御できる以外に、ディスパッチキューでデータを非同期に処理できるためです。ディスパッチI/Oチャンネルは、読み書きの必要な内容のファイルを識別する`dispatch_io_t`構造です。ファイルのストリームベースのアクセスまたはランダムアクセスを、チャンネルに設定できます。ストリームベースのチャンネルは、ファイルデータを連続的に読み書きするのに対し、ランダムアクセスチャンネルではファイルの先頭からの任意のオフセットで読み書きできます。

ディスパッチI/Oチャンネルの作成と管理で問題が発生するのを防ぐ場合、`dispatch_read(3)` Mac OS X Developer Tools Manual Pageまたは`dispatch_write(3)` Mac OS X Developer Tools Manual Page関数を使用すると、ファイル記述子で単一の読み書き操作を実行できます。これらのメソッドは、ディスパッチI/Oチャンネルの作成と管理でオーバーヘッドの発生を防ぐ状況に適しています。た

だし、前記のメソッドは、ファイルで単一の読み書き操作を実行する場合にのみ使用してください。同じファイルで複数の操作を実行する必要がある場合、ディスパッチI/Oチャンネルを作成する方が効率的です。

ディスパッチソースを使用すると、Cocoaストリームオブジェクトと似た方法でファイルを処理できます。ストリームオブジェクトと同様に、ディスパッチソースは多くの場合、ソケットまたはデータを突発的に送受信するデータソースで使用されますが、ファイルにも使用できます。ディスパッチソースは、読み取りを待機しているデータがある場合、または書き込みに使用できるスペースがある場合は常に、対応するイベントハンドラブロックをスケジュールします。ファイルの場合、通常はディスパッチソースを明示的にキャンセルするまで、または読み取り中のファイルの末尾に到達するまで、ブロックが反復的に、また短時間に連続的にスケジュールされます。

ディスパッチソースを作成し、使用方法については、『*Concurrency Programming Guide*』を参照してください。dispatch_read(3) Mac OS X Developer Tools Manual Pageまたは dispatch_write(3) Mac OS X Developer Tools Manual Page関数、またはその他のGCD関数の詳細については、『*Grand Central Dispatch (GCD) Reference*』を参照してください。

ディスパッチI/Oチャンネルの作成と使用

ディスパッチI/Oチャンネルを作成する場合、ファイル記述子かまたは開こうとするファイルの名前を指定する必要があります。開いているファイルの記述子をすでに指定している場合、チャンネルに渡すことでファイル記述子の所有権がコードからチャンネルに移行します。ディスパッチI/Oチャンネルはファイル記述子を制御するため、必要に応じてファイル記述子を再設定できます。たとえば、チャンネルは通常はO_NONBLOCKフラグを使ってファイル記述子を再設定するため、以降の読み書き操作により現在のスレッドがブロックされません。ファイルパスを使用してチャンネルを作成した場合、必要なファイル記述子がチャンネルで作成され、チャンネルがこの記述子を制御します。

リスト 7-1に、NSURLオブジェクトを使用してディスパッチI/Oチャンネルを開く方法の簡単な例を示します。この例では、チャンネルはランダム読み取りアクセスで設定され、現在のクラスのカスタムプロパティに割り当てられます。キューとブロックは、作成時にエラーが発生した場合、またはチャンネルのライフサイクルの終了時に動作し、チャンネルをクリーンアップします。作成中にエラーが起こった場合、エラーコードを使用して発生した内容を判断できます。エラーコード0は、通常は dispatch_io_close(3) Mac OS X Developer Tools Manual Page関数の呼び出しの結果、チャンネルがファイル記述子の制御を正常に放棄したこと、およびチャンネルを安全に破棄できることを示します。

リスト 7-1 ディスパッチI/Oチャンネルの作成

```
-(void)openChannelWithURL:(NSURL*)anURL {
    NSString* filePath = [anURL path];
    self.channel = dispatch_io_create_with_path(DISPATCH_IO_RANDOM,
```

```
    [filePath UTF8String],    // C文字列への変換
    O_RDONLY,                // 読み取り用に開く
    0,                      // 追加フラグなし
    dispatch_get_main_queue(),
    ^(int error){
        // 通常のチャンネル操作のためのコードをクリーンアップ
        // dispatch_io_closeがほかの場所で呼び出されたと仮定する
        if (error == 0) {
            dispatch_release(self.channel);
            self.channel = NULL;
        }
    });
}
```

ディスパッチチャンネルを作成した後、作成後のdispatch_io_t構造の参照を保存し、必要に応じて読み書きの呼び出しを起動するときに使用できます。ランダムアクセスをサポートするチャンネルを作成している場合、任意の場所で読み書きを開始できます。ストリームベースのチャンネルを作成する場合、開始点として指定する任意のオフセット値が無視され、データは現在の場所から読み書きされます。たとえば、ランダムアクセスをサポートするチャンネルの2番目の1024バイトを読み取る場合、読み取り呼び出しは以下のようになります。

```
dispatch_io_read(self.channel,
    1024,                // ファイル内の1024バイト
    1024,                // 次の1024バイトを読み取る
    dispatch_get_main_queue(), // メインスレッドでバイトを処理する
    ^(bool done, dispatch_data_t data, int error){
        if (error == 0) {
            // バイトを処理する
        }
    });
```

書き込み操作では、ファイルへの書き込みが必要なバイト、（ランダムアクセスチャンネルの）書き込みを開始する場所、および進行状況の報告を受信するハンドラブロックを指定する必要があります。書き込み操作は、*Grand Central Dispatch (GCD) Reference*で説明するdispatch_io_write(3) Mac OS X Developer Tools Manual Page関数を使用して開始します。

I/Oチャンネルのディスパッチデータの操作

すべてのチャンネルベースの操作は、`dispatch_data_t`構造を使用してチャンネルを使用したデータの読み書きを操作します。`dispatch_data_t`構造は、1つまたは複数の連続的なメモリバッファを管理する不透過型です。不透過型の使用により、GCDはほぼ連続的にアプリケーションにデータを提供しながら、不連続のバッファを内部的に使用できます。実際に実装される、ディスパッチデータ構造の機能の詳細は重要ではありませんが、同構造の作成方法やデータの取得方法を理解することは重要です。

ディスパッチI/Oチャンネルにデータを書き込む場合、コードにより`dispatch_data_t`構造と書き込むバイト数を指定する必要があります。この操作は`dispatch_data_create(3)` Mac OS X Developer Tools Manual Page関数を使用して行います。この関数はバッファのポインタとこのバッファのサイズを取り出し、バッファのデータを列挙する`dispatch_data_t`構造を返します。データオブジェクトがバッファをカプセル化する方法は、`dispatch_data_create(3)` Mac OS X Developer Tools Manual Page関数を呼び出すときに指定するデストラクタに依存します。デフォルトのデストラクタを使用する場合、データオブジェクトはバッファのコピーを作成し、そのバッファを適時解放する役目を担います。ただし、指定したバッファをデータオブジェクトを使ってコピーしない場合、データオブジェクトが解放されたときに必要なクリーンアップを処理するカスタムデストラクタブロックを指定する必要があります。

注意 単一の連続したデータブロックとしてファイルに書き込むデータバッファが複数存在する場合、すべてのバッファを表す単一のディスパッチデータオブジェクトを作成できません。`dispatch_data_create_concat(3)` Mac OS X Developer Tools Manual Page関数を使用して、追加データバッファを`dispatch_data_t`構造に付加します。バッファ自体はすべて独立し、メモリの異なる箇所に置かれていますが、ディスパッチデータオブジェクトはバッファを収集し、単一のエンティティとして表示します

(`dispatch_data_create_map(3)` Mac OS X Developer Tools Manual Page関数は、連続した版のバッファを生成する場合にも使用できます)。特にディスクベースの操作の場合、複数のバッファの連結により、`dispatch_io_write(3)` Mac OS X Developer Tools Manual Page関数を1度呼び出すだけで大量のデータをファイルに書き込むことができます。これは独立したバッファで個別に`dispatch_io_write(3)` Mac OS X Developer Tools Manual Pageを呼び出すよりも、かなり効率的です。

ディスパッチデータオブジェクトからバイトを抽出する場合、`dispatch_data_apply(3)` Mac OS X Developer Tools Manual Page関数を使用します。ディスパッチデータオブジェクトは不透過であるため、オブジェクトの各バッファでこの関数を反復して使用し、指定したブロックを使用してバッファを処理します。連続した単一のバッファを使用するディスパッチデータオブジェクトの場合、ブ

ロックは1度呼び出されます。複数のバッファを使用するデータオブジェクトの場合、ブロックはバッファの数だけ呼び出されます。ブロックが呼び出されると毎回、データバッファと、そのバッファに関する何らかの情報が渡されます。

リスト 7-2に、ファイルの内容のNSStringオブジェクトを作成し、チャンネルを開き、UTF8形式のテキストファイルを読み取る例を示します。この特別な例では、1度に任意の量である1024バイトが読み取られていますが、最適なパフォーマンスが得られていない可能性があります。ただし、dispatch_io_read(3) Mac OS X Developer Tools Manual Page関数をdispatch_data_apply(3) Mac OS X Developer Tools Manual Page関数と併用し、バイトを読み取り、アプリケーションで要求される形式にバイトを変換するプロセスという基本的な前提が実証されています。この場合、バイトを処理するブロックは、ディスパッチデータオブジェクトのバッファを使用して、新しい文字列オブジェクトを初期化します。次に文字列をカスタムaddStringToFile:メソッドにハンドオフします。この場合、このメソッドは文字列を保持し、後でできるように保存します。

リスト 7-2 ディスパッチI/Oチャンネルを使用したテキストファイルからのバイトの読み取り

```
- (void)readContentsOfFile:(NSURL*)anURL {
    // 読み取るチャンネルを開く
    NSString* filePath = [anURL path];
    self.channel = dispatch_io_create_with_path(DISPATCH_IO_RANDOM,
        [filePath UTF8String], // C文字列への変換
        O_RDONLY,             // 読み取り用に開く
        0,                     // 追加フラグなし
        dispatch_get_main_queue(),
        ^(int error){
            // コードをクリーンアップ
            if (error == 0) {
                dispatch_release(self.channel);
                self.channel = nil;
            }
        });

    // ファイルチャンネルを作成できない場合、中止する
    if (!self.channel)
        return;

    // ファイルのサイズを取得する
    NSNumber* theSize;
```

```
NSInteger fileSize = 0;
if ([anURL getResourceValue:&theSize forKey:NSURLFileSizeKey error:nil])
    fileSize = [theSize integerValue];

// ファイルを1024サイズの文字列に分割する
size_t chunkSize = 1024;
off_t currentOffset = 0;

for (currentOffset = 0; currentOffset < fileSize; currentOffset += chunkSize)
{
    dispatch_io_read(self.channel, currentOffset, chunkSize,
dispatch_get_main_queue(),
        ^(bool done, dispatch_data_t data, int error){
            if (error)
                return;

            // データから文字列を構築する
            dispatch_data_apply(data,
(dispatch_data_applier_t)^(dispatch_data_t region,
                                size_t offset, const void *buffer,
                                size_t size){
                NSAutoreleasePool* pool = [[NSAutoreleasePool alloc]
init];

                NSString* aString = [[[NSString alloc]
initWithBytes:buffer
                                length:size encoding:NSUTF8StringEncoding]
autorelease];

                [self addString:aString toFile:anURL]; // カスタムメソッ
ド

                [pool release];
                return true; // データがほかにある場合、処理を継続する
            });
        });
}
```

ディスパッチデータオブジェクトの操作に使用する関数の詳細については、『*Grand Central Dispatch (GCD) Reference*』を参照してください。

ファイルの同期的な読み書き

データを同期的に操作するファイル関連のインターフェイスでは、コードの実行コンテキストを自分で柔軟に設定できます。同期的な実行というだけで、非同期的な実行よりも非効率的になるという意味ではありません。単純に、このインターフェイスでは、非同期の実行コンテキストが自動的に提供されないということです。上記の技法を使用してデータを非同期的に読み書きし、同じ結果を得る場合は、非同期実行コンテキストを自分で指定する必要があります。最善の方法は、当然ですが、GCD ディスパッチキューまたは操作オブジェクトを使用してコードを実行する方法です。

メモリの内容の構築とディスクへの書き込み操作の同時実施

ファイルデータの読み書きを管理するもっとも簡単な方法は、同時実行です。この方法が適するのは、ドキュメントのディスク上の表示サイズが比較的少ないと予測される、カスタムドキュメントタイプです。マルチメディアファイルや、サイズがメガバイトレベルまで増大するファイルにはこの方法は実行できない場合があります。

バイナリファイル形式または非公開ファイル形式を使用するカスタムドキュメントタイプの場合、NSDataまたはNSMutableDataクラスを使用してディスク間でカスタムデータをやり取りできます。新しいデータオブジェクトは、多くの異なる方法で作成できます。たとえば、キー付きアーカイバオブジェクトを使用すると、オブジェクトのグラフを、データオブジェクトで囲まれたバイトの線形ストリームに変換することができます。バイナリファイル形式が複雑に構造化されている場合、NSMutableDataオブジェクトにバイトを付加し、データオブジェクトを断片的に構築できます。データオブジェクトのディスクへの書き込み準備が整っている場合、writeToURL:atomically:またはwriteToURL:options:error:メソッドを使用します。これらのメソッドにより、対応するディスク上のファイルを1ステップで作成できます。

注意 iOSでは、writeToURL:options:error:メソッドに渡すオプションの1つを使って、ファイルの内容を暗号化するかどうかを指定できます。これらのオプションの1つを指定した場合、内容は即時暗号化され、ファイルのセキュリティが確保されます。

元のディスクからデータを読み取る場合、initWithContentsOfURL:options:error:メソッドを使用して、ファイルの内容に基づくデータオブジェクトを取得します。このデータオブジェクトは、作成時に用いたプロセスを逆転する場合に使用できます。この場合、キー付きアーカイバを使用してデータオブジェクトを作成している場合、オブジェクトグラフの再作成にキー付きアンアーカイバを使用できます。データを断片的に書き出ししている場合、データオブジェクトでバイトストリームを解析し、ドキュメントのデータ構造の再構築に使用できます。

NSDocumentインフラストラクチャを使用するアプリケーションは、通常はNSDataオブジェクトを使用して間接的にファイルシステムと対話します。ユーザがドキュメントを保存する場合、インフラストラクチャからデータオブジェクトの対応するNSDocumentオブジェクトに対して、ディスクへの書き込みが要求されます。同様に、ユーザが既存のドキュメントを開く場合、ドキュメントオブジェクトを作成し、初期化するデータオブジェクトに渡します。

NSDataクラスとNSMutableDataクラスの詳細については、『*Foundation Framework Reference*』を参照してください。OS Xアプリケーションのドキュメントインフラストラクチャの使い方については、『*Mac App Programming Guide*』を参照してください。

NSFileHandleを使用したファイルの読み書き

NSFileHandleクラスの使用は、POSIXレベルでファイルを読み書きするプロセスに非常に似ています。基本的に、ファイルを開き、読み書きの呼び出しを発行し、終了後にファイルを閉じるというプロセスをとります。NSFileHandleの場合、クラスのインスタンスを作成すると自動的にファイルが表示されます。ファイル処理オブジェクトは、ファイルのラップアとして動作し、基本ファイル記述子を管理します。読み取りアクセス、書き込みアクセス、または両方のアクセスのどのアクセスを要求しているかに応じて、NSFileHandleのメソッドを呼び出し、実際のバイトを読み書きします。処理が終了したら、ファイル処理オブジェクトを解放し、ファイルを閉じます。

リスト 7-3に、ファイル処理オブジェクトを使用してファイルの内容全体を読み取る、非常に簡単なメソッドを示します。fileHandleForReadingFromURL:error:メソッドは、ファイル処理オブジェクトを自動解放されたオブジェクトとして作成し、このメソッドが返された後のある時点で自動的に解放されます。

リスト 7-3 NSFileHandleを使用したファイルの内容の読み取り

```
- (NSData*)readDataFromFileAtURL:(NSURL*)anURL {
    NSFileHandle* aHandle = [NSFileHandle fileHandleForReadingFromURL:anURL
error:nil];
    NSData* fileContents = nil;

    if (aHandle)
        fileContents = [aHandle readDataToEndOfFile];

    return fileContents;
}
```

NSFileHandleクラスのメソッドの詳細については、『*NSFileHandle Class Reference*』を参照してください。

ディスクの読み書きのPOSIXレベルの管理

ファイル管理コードにCベースの関数を多用している場合、POSIXレイヤはファイル进行操作するための標準関数を提供します。POSIXレベルでは、アプリケーション内で開かれたファイルを一意に識別する整数値である、ファイル記述子を使ってファイルを特定します。このファイル記述子を、同記述子を必要とする後続の関数に渡します。以下のリストには、ファイルの操作に使用する主なPOSIX関数を示しています。

- `open`関数は、ファイルのファイル記述子の取得に使用します。
- `pread`、`read`、または`readv`関数は、開かれたファイル記述子からデータを読み取る場合に使用します。上記の関数の詳細については、『`pread`』を参照してください。
- `pwrite`、`write`、または`writenv`関数は、開かれたファイル記述子にデータを書き込む場合に使用します。上記の関数の詳細については、『`pwrite`』を参照してください。
- `lseek`関数は、現在のファイルポインタの再配置、およびデータを読み書きする場所の変更に使用します。
- `pclose`関数は、ファイル記述子の操作の終了時に、この記述子を閉じる場合に使用します。

Important 一部のファイルシステムでは、書き込み呼び出しの成功により、バイトが実際にファイルシステムに書き込まれるかどうかは保証されません。一部のネットワークファイルシステムについては、書き込まれたデータは書き込み呼び出しの後のある時点で、サーバに送信される場合があります。データが実際にファイルに書き込まれたことを確認する場合、`fsync`関数を使用して、強制的にサーバにデータを送信するか、ファイルを閉じて、正常に終了するかどうかを確認します。すなわち`close`関数が-1を返さないことを確認します。

リスト7-4に、POSIX呼び出しを使用してファイルの最初の1024バイトを読み取り、このバイトをNSDataオブジェクトで返す簡単な関数を示します。ファイルのサイズが1024バイトを下回る場合、このメソッドは可能な限り多くのバイトを読み取り、データオブジェクトを実際のオブジェクト数まで切り詰めます。

リスト 7-4 POSIX関数を使用したファイルの内容の読み取り

```
- (NSData*)readDataFromFileAtURL:(NSURL*)anURL {
    NSString* filePath = [anURL path];
    fd = open([filePath UTF8String], O_RDONLY);
    if (fd == -1)
```

```
        return nil;

    NSMutableData* theData = [[[NSMutableData alloc] initWithLength:1024]
autorelease];
    if (theData) {
        void* buffer = [theData mutableBytes];
        NSUInteger bufferSize = [theData length];

        NSUInteger actualBytes = read(fd, buffer, bufferSize);
        if (actualBytes < 1024)
            [theData setLength:actualBytes];
    }

    close(fd);
    return theData;
}
```

アプリケーションが所定の時間に開くファイル記述子の数には制限があるため、ファイル記述子の使用が終了したら速やかに同記述子を閉じるようにしてください。ファイル記述子は、開かれたファイル以外に、ソケットやパイプなどの通信チャンネルにも使用されます。またコード以外にも、アプリケーションにファイル記述子を作成するエンティティがあります。リソースファイルをロードする場合、またはネットワークで通信するフレームワークを使用する場合は毎回、コードに代わってシステムでファイル記述子が作成されます。コードが大量のソケットまたはファイルを開き、閉じない場合、システムのフレームワークは厳密なタイミングでファイル記述子を作成できない場合があります。

ファイルメタデータ情報の取得と設定

ファイルには大量の有用な情報が含まれますが、ファイルシステムも同様です。各ファイルおよびディレクトリに対して、ファイルシステムは項目のサイズ、作成日、オーナー、権限、ファイルのロックの有無、ファイルの拡張子の表示または非表示などのメタ情報を保存します。この情報を取得し、設定する方法は複数ありますが、もっとも一般的な方法を以下に示します。

- 内容のメタデータ情報のほとんど（アプリケーション固有の属性を含む）は、`NSURL`クラスを使用して取得および設定します。
- 基本的なファイル関連の情報は、`NSFileManager`クラスを使用して取得および設定します。

NSURLクラスは、ファイルシステムの標準情報（ファイルサイズ、タイプ、オーナー、権限など）以外にApple固有の多くの情報（割り当てられたレベル、ローカライズ名、ファイルに関連付けられたアイコン、ファイルがパッケージか否かなど）を含む、ファイル関連の幅広い情報を提供します。さらに、URLを引数に取る一部のメソッドを使用すると、ファイルまたはディレクトリで別の操作を行っている間に、属性をキャッシュすることができます。特に大量のファイルにアクセスする場合、このタイプのキャッシュ動作はディスク関連の操作数が最小限に抑えられ、パフォーマンスを向上させる可能性があります。属性がキャッシュされているかどうかに関わらず、NSURLオブジェクトの `getResourceValue:forKey:error:` メソッドを使用して属性を取得し、一部の属性の新しい値は `setResourceValue:forKey:error:` または `setResourceValues:error:` メソッドを使用して設定します。

NSURLクラスを使用しない場合においても、一部のファイル関連情報の取得と設定はNSFileManagerクラスの `attributesOfItemAtPath:error:` メソッドと `setAttributes:ofItemAtPath:error:` メソッドを使用して実行できます。これらのメソッドを使用すると、タイプやサイズ、サポートされているアクセスレベルなどの、ファイルシステムの項目に関する情報を取得できます。またファイルシステム自体に関する一般的な情報、すなわちサイズや空きスペース量、ファイルシステムのノード数などの取得にはNSFileManagerクラスも使用できます。iCloudファイルについては、リソースのサブセットのみ取得できることに注意してください。

NSURLクラスとNSFileManagerクラス、およびファイルとディレクトリの取得可能な属性の詳細については、『*NSURL Class Reference*』と『*NSFileManager Class Reference*』を参照してください。

ファイルラッパーのファイルコンテナとしての使用

NSFileWrapperインスタンスは、ファイルの内容を動的メモリ内に保持します。同インスタンスのこの役割により、ドキュメントオブジェクトにファイルを埋め込むことができ、画像としての表示（および表示状態での編集）、ディスクへの保存、または別のアプリケーションへの転送が可能なデータ単位としてファイル进行处理します。このインスタンスはまた、ドキュメントまたはドラッグ操作でファイルを表すアイコンを保存することもできます。

このクラスのインスタンスは、ファイルラッパーオブジェクトと呼ばれ、また混乱が起これなければ単にファイルラッパーと呼ばれます。ファイルラッパーは、3種類のラッパー、すなわち実際の1つのファイルの内容を保持する通常ファイルラッパー、ディレクトリとその内部のすべてのファイルまたはディレクトリを保持するディレクトリラッパー、ファイルシステムのシンボリックリンクのみを表すリンクラッパーのいずれかになります。

ファイルラッパーの目的はメモリ内のファイルを表すことであるため、どのディスク表現との連結も非常に緩やかです。ファイルラッパーは、ディスク内容の表現のパスを記録しません。このため、異なるURLで同じファイルラッパーを保存できますが、後でディスクのファイルラッパーを更新する必要がある場合は、そのURLの記録が要求されます。

注意 NSFileWrapperインスタンスがファイル調整のための項目として指定される場合、ファイルラッパー内のすべてのファイルは自動的にそのファイル調整の対象となります。各ファイルまたはディレクトリを個別に管理する必要はありません。

ファイルラッパーの操作

ファイルラッパーは、initWithSerializedRepresentation:メソッドを使用してメモリのデータから、またはinitWithURL:options:error:メソッドを使用してディスクのデータから作成できます。いずれも、直列化表示またはディスク上のファイルの性質に基づいて、適切なタイプのファイルラッパーが作成されます。

3つの簡易メソッド、initWithRegularFileWithContents:, initWithDirectoryWithFileWrappers:, initWithSymbolicLinkWithDestination:はそれぞれ特定のタイプのファイルラッパーを生成します。それぞれの初期化メソッドからタイプまたは状態の異なるファイルラッパーが生成されるため、すべてのメソッドにこのクラスのイニシャライザが指定されており、必要に応じてサブクラスがすべてオーバーライドしなければなりません。

一部のファイルラッパーメソッドは特定のラッパータイプにのみ適用され、間違っただタイプのファイルラッパーにメソッドが送信された場合、例外が発生します。ファイルラッパーのタイプを判断する場合は、`isRegularFile`、`isDirectory`、`isSymbolicLink`メソッドを使用します。

ファイルラッパーはファイルシステム情報（変更時間とアクセス権など）を保存し、この情報はディスクから読み取られた場合に更新され、ディスクにファイルを書き込むときに使用されます。`fileAttributes`メソッドは、`NSFileManager`メソッドの`attributesOfItemAtPath:error:`で説明する形式でこの情報を返します。またファイルの属性を`setFileAttributes:`メソッドを使用して設定することもできます。

`NSFileWrapper`クラスを使用すると、保存操作に適切なファイル名を設定することができます。このクラスは実際に保存された最後のファイル名を記録し、`preferredFilename`と`filename`メソッドはこれらの名前を返します。この機能はディレクトリラッパーでもっとも重要ですが、“ディレクトリラッパーの操作”内でも同様の内容が説明されています。

ファイルラッパーをディスクに保存する場合、通常は保存に必要なディレクトリを決定し、そのディレクトリURLに適切なファイル名を付加し、`writeToURL:options:originalContentsURL:error:`メソッドを使用します。この操作によりファイルラッパーの内容が保存され、ファイルの属性が更新されます。必要に応じて、ファイルラッパーと別の名前で保存できますが、この場合、`writeToURL:options:originalContentsURL:error:`メソッドの呼び出し方に応じて、記録されたファイル名が適切なファイル名と異なる結果になる場合があります。

ファイルラッパーは、その内容をディスクに保存する以外に、その内容を必要に応じてディスクから読み取ることができます。`matchesContentsOfURL:`メソッドは、ファイルが最後に読み取られたとき、または書き込まれたときに保存されたファイルの属性に基づき、ディスク表現が変更されているかどうかを判断します。ファイルラッパーの変更時間またはアクセス権がディスク上のファイルの内容と異なる場合、このメソッドはYESを返します。この後、`readFromURL:options:error:`を使用して、ディスクからファイルを再度読み取ります。

最後に、（ペーストボードなどを使用して）別のプロセスまたはシステムにファイルラッパーを転送する場合、`serializedRepresentation`メソッドを使用して、ファイルラッパーの内容を`NSFileContentsPboardType`形式で格納した`NSData`オブジェクトを取得します。この表現はどのチャネルを選択した場合でも安全に転送できます。表現の受領側は、この後、`initWithSerializedRepresentation:`メソッドを使用してファイルラッパーを再構築できます。

ディレクトリラッパーの操作

ディレクトリラッパーは、（任意のタイプの）ほかのファイルラッパーを含み、適切なファイル名から導き出されたキーを使用してアクセスできます。任意のタイプのファイルラッパーを`addFileWrapper:`メソッドを使ってディレクトリに追加し、`removeFileWrapper:`メソッドを使って

ディレクトリから削除します。簡易メソッド、`addRegularFileWithContents:preferredFilename:`と`addSymbolicLinkWithDestination:preferredFilename:`を使用すると、適切な名前を設定しながら、通常のファイルとリンクのラッパーを追加できます。

ディレクトリラッパーは、その内容を`NSDictionary`オブジェクトに保存し、これは`fileWrappers`メソッドを使用して取得します。この辞書のキーは、ディレクトリラッパー内の各ファイルラッパーの適切なファイル名に基づきます。ディレクトリラッパー内のファイルラッパーには、3つの識別子が存在します。

- **適切なファイル名。**この識別子は、ファイルラッパーを一意に特定しませんが、ほかの識別子は常にこの識別子に基づきます。
- **辞書キー。**この識別子は、同じディレクトリラッパーに同じ適切な名前のファイルラッパーがほかに存在しない場合、この適切な名前に等しくなります。それ以外の場合、適切なファイル名に固有のプレフィックスを追加して生成された文字列になります。同じファイルラッパーが、所属するディレクトリラッパーごとに異なる辞書キーを持つ場合があることに注意してください。辞書キーは、メモリ内のラッパーオブジェクトの内容またはファイル名を取得するために（すなわち、ディスクから更新するために）、このオブジェクトの取得に使用します。ファイルラッパーの辞書キーは、所属するディレクトリラッパーに`keyForFileWrapper:`メッセージを送信すると取得できます。
- **ファイル名。**この識別子は、通常は適切なファイル名に基づきますが、適切なファイル名あるいは辞書キーと同じになるとは限りません。ファイル名は、所属するディレクトリラッパーのパスを基準とした、単一のファイルラッパーの更新に使用します。ファイルラッパーを含むディレクトリラッパーを保存した場合は常に、ファイル名が変更されることに注意してください。特に、ファイルラッパーが複数の異なるディレクトリラッパーに追加されている場合です。このため、ファイル名が必要な場合は必ずファイルラッパーから取得し、ファイル名をキャッシュしないでください。

ディレクトリラッパーの内容を操作する場合、辞書列挙子を使用すると、各ファイルラッパーを取得し、必要なすべての操作を実行できます。保存と更新を例外として、ディレクトリファイルラッパーはその内容に対して再帰的な操作を定義しません。所属するすべてのファイルラッパーにファイル属性を設定する場合、あるいは同様のほかの操作を実行する場合、各ファイルラッパーのタイプを検証し、ディレクトリラッパーが発生するたびに反復的に起動する再帰的メソッドを定義しなければなりません。

パフォーマンスのヒント

アプリケーションが大量のファイルを処理する場合、そのファイル関連のコードのパフォーマンスがきわめて重要になります。ほかのタイプの操作と比較すると、ディスク上のファイルへのアクセスは、もっともコンピュータの動作を遅くする操作の1つです。ファイルのサイズと数に応じて、ディスクベースのハードドライブからファイルを読み取る時間は、数ミリ秒から数分までの範囲です。したがって、軽度から中程度の作業負荷においても、コードに最大限のパフォーマンスを確実に発揮させる必要があります。

アプリケーションがファイル処理の際に、低速化または応答への遅延を発生する場合、Instrumentsアプリケーションを使用していくつかの基準を収集します。Instrumentsは、アプリケーションがファイル処理に費やした時間を示し、さまざまなファイル関連のアクティビティの監視をサポートします。それぞれの問題を修正した後、必ず再度Instrumentsでコードを実行し、結果を記録し、実施した変更が効果を上げているかどうかを確認できるようにしてください。

コード内の検出項目

ファイル関連のコードの潜在的な修正箇所がわからない場合は、下記のヒントを参照してください。

- **コードがディスクから（任意のタイプの）大量のファイルを読み取る箇所を調べます。** リソースファイルをロードしている箇所も、忘れずに調べてください。すべてのファイルのデータを直ちに使用する予定がありますか? 使用しない場合、ファイルの一部のロードを後回しにしても構いません。
- **古いファイルシステムの呼び出しを使用している箇所を調べます。** 呼び出しのほとんどは、Objective-Cインターフェイスまたはブロックベースのインターフェイスを使用します。BSDレベルの呼び出しも使用できますが、FSRefまたはFSSpecデータ構造で動作するCarbonベースの古い関数を使用することはできません。Xcodeは、廃止されたメソッドおよび関数を使用するコードを検出すると警告を生成するため、それらの警告を確認する必要があります。
- **コールバック関数またはメソッドを使用して、ファイルデータを処理している箇所を調べます。** ブロックオブジェクトを使用する新しいAPIが利用できる場合、そのAPIを使用できるようにコードを更新することをお勧めします。
- **負荷の少ない読み取り操作を数多く同じファイルで実行している箇所を調べます。** それらの操作をまとめて、一度に実行できませんか? 同じ量のデータであれば、通常は負荷の少ない操作を多く実行するよりも、1つの大きな読み書き操作を実行する方が効率的です。

最新のファイルシステムインターフェイスの使用

呼び出すルーチンを決定する場合、文字列を使用してパスを指定するルーチンよりも、NSURLオブジェクトを使用してパスを指定できるルーチンを優先的に選択してください。URLベースのルーチンのほとんどは、OS X 10.6以降で導入され、始めからGrand Central Dispatchなどの技術が利用できるように設計されています。このため、多くの介入を必要とせずに、コード内で即座にマルチコアコンピュータの効果を実現できます。

またコールバック関数またはメソッドを受け付けるルーチンよりも、ブロックオブジェクトを受け付けるルーチンを優先する必要があります。ブロックは、コールバックタイプの動作を実装する便利で効率的な手段です。実際に多くの場合、ブロックは実装のために必要なコードが多くありません。ブロックは、データを受け渡すためのコンテキストデータ構造の定義と管理を要求しないためです。GCDキューを使ったスケジューリングでブロックを実行するルーチンもあり、この場合もパフォーマンスが改善されます。

一般的なヒント

以下に示すのは、プログラムのI/Oアクティビティを減らすための基本的な推奨事項です。ファイルシステムに関連したパフォーマンスの改善に役立つ場合がありますが、ほかのヒントと同様に、改善度を検証できるように、必ず前後のパフォーマンスを測定してください。

- **実行するファイル操作の数を最小限に抑えてください。** ローカルファイルシステムのデータをメモリに移動する処理には、相当の時間を要します。ファイルシステムのアクセス回数は一般にミリ秒で測定され、データのディスクからの読み込みを待機するための、数100万回のクロックサイクルに相当します。またターゲットファイルシステムが地球の反対側のサーバに存在する場合、ネットワークの遅延によりデータの取得が遅れます。
- **パスオブジェクトを再利用します。** 時間をかけてファイルにNSURLを作成している場合、必要に応じて作成するのではなく、同じオブジェクトを最大限再利用してください。ファイルの検索とURLまたはパス名情報の構築には時間がかかり、コスト高になる場合もあります。そのような操作から作成したオブジェクトを再利用することで、時間が節約され、アプリケーションのファイルシステムとの相互作用も最小限に抑えられます。
- **適切なサイズの読み取りバッファを選択します。** ディスクからローカルバッファにデータを読み取る場合、選択するバッファのサイズは操作の速度に大きく影響する場合があります。比較的容量の大きなファイルを操作する場合、1Kのバッファを読み取りに割り当て、データを小さな単位で処理するのは理にかないません。この場合は、大容量のバッファ（128Kから256Kのサイズ）を作成し、データの大部分または全データをメモリに読み取ってから処理します。ディスクへのデータの書き込みにも同じ法則が適用されます。1度のファイルシステムの呼び出しで、可能な限り連続的にデータを書き込みます。

- **ファイル内をジャンプで移動するのではなく、データを連続的に読み取ります。**カーネルは透過的にI/O操作をまとめ、連続的な読み取りをさらに高速化します。
- **データの書き込みの前の、空ファイル内での先飛ばしを禁止します。**システムでは、間のスペースにゼロを書き込んで隙間が埋められます。書き込み時にファイルに「穴」を含めるのは相応の理由がありますが、その方法ではパフォーマンスに影響する場合があることを覚えておいてください。詳細については、“[Zero-Fill遅延によるセキュリティの強化と高コスト化](#)”（96 ページ）を参照してください。
- **アプリケーションがデータを必要とするまで、I/O操作を延期します。**遅延という黄金の法則は、ディスクのパフォーマンスとその他の多くのパフォーマンスにも適用されます。
- **環境設定システムを不正に使用しないでください。**環境設定システムは、低コストで再計算が可能なデータではなく、ユーザの環境設定（ウィンドウ位置、ビュー設定、ユーザ指定の環境設定など）の取り込みにのみ使用します。簡単な値の再計算は、同じ値をディスクから読み取るよりも相当に速くなります。
- **ファイルのメモリへのキャッシュによりアプリケーションが高速化すると想定しないでください。**ファイルのメモリへのキャッシュはメモリ利用率を高め、別の意味でパフォーマンスを低下させる可能性があります。さらに、システムは一部のファイルデータを自動的にキャッシュするため、独自にキャッシュを作成するとさらに状況が悪化する場合があります。“[システム独自のファイルキャッシュメカニズム](#)”（95 ページ）を参照してください。

システム独自のファイルキャッシュメカニズム

ディスクへのキャッシュはファイルデータへのアクセスを効率化する適切な方法ですが、すべての状況で適切な用途とは限りません。キャッシュはアプリケーションのメモリ占有量を高め、適切に使用していなければ、ディスクからデータを再ロードするよりも高コストになる場合もあります。

キャッシュは複数回のアクセスを予定しているファイルにはもっとも適した方法です。ファイルの使用が1回限りであれば、キャッシュを無効にするか、ファイルをメモリにマッピングすべきです。

ファイルシステムのキャッシュの無効化

大容量のマルチメディアファイルのストリーミングなど、当座は再度利用しないことが確実なデータを読み取る場合、そのデータをファイルシステムのキャッシュに追加しないようにファイルシステムに指示します。デフォルトでは、システムのバッファキャッシュは、最後にディスクから読み取られたデータを保持されます。このディスクキャッシュは、頻繁に使用されるデータを格納しているときにもっとも効果を現します。ファイルのキャッシュを有効にしたまま、大容量のマルチメディアファイルをストリーミングした場合、再度使用しないデータでディスクのキャッシュがすぐに一杯になってしまいます。それ以外に、このプロセスによりキャッシュからほかのデータが押し出されてしまう可能性があり、そのデータはキャッシュ内で得られたはずの便益を失うことになります。

ファイルのキャッシュを有効または無効にする場合、アプリケーションはBSDの`fcntl`関数を`F_NOCACHE`フラグで呼び出します。この関数の詳細については、『`fcntl`』を参照してください。

注意 キャッシュされていないデータを読み取る場合、4Kアライメントバッファの使用が推奨されます。このバッファは、データのメモリへのロードを柔軟なものにし、最終的にロード時間を短縮します。

キャッシュの代用としてのI/Oマッピング

ファイルからデータをランダムに読み取る予定であれば、ファイルを直接アプリケーションの仮想メモリスペースにマッピングする方法で、パフォーマンスを改善できる状況があります。ファイルのマッピングは、読み取り専用の権利でファイルにアクセスする場合に便利なプログラミングです。カーネルによる仮想メモリページングメカニズムの活用が可能になり、ファイルデータは必要なときにのみ読み取られます。またファイルのマッピングを使用して、ファイル内の既存のバイトをオーバーライドできます。ただし、この方法でファイルのサイズを拡大することはできません。マッピングされたファイルはシステムのディスクキャッシュをバイパスするため、ファイルの1コピーのみがメモリに保存されます。

Important ファイルをメモリにマップし、ファイルへのアクセスが不可になった場合、これはファイルを含むディスクがイジェクトされているか、ファイルを含むネットワークサーバがマウントされていないことが理由ですが、アプリケーションはクラッシュしてSIGBUSエラーを発行します。

ファイルをメモリにマッピングする方法については、『*File System Advanced Programming Topics*』を参照してください。

Zero-Fill遅延によるセキュリティの強化と高コスト化

セキュリティ上の理由により、ディスク上の区画がファイルに割り当てられると、ファイルシステムによりゼロアウトされると想定されます。この動作により、削除済みのファイルの残りのデータが、新しいファイルにインクルードされることが解消されます。OS Xで使用されるHFS Plusファイルシステムは、常にこのZero-fill動作を実装しています。

読み取りと書き込みのいずれの操作についても、システムは最後の瞬間までゼロの書き込みを遅らせます。書き込みの後、ファイルを閉じると、コードがアクセスしなかったファイルの部分にシステムがゼロを書き込みます。ファイルから読み取る場合、システムが新しい区画にゼロを書き込むのは、コードがその区画からの読み取りを試みた場合、またはファイルが閉じられる場合に限られます。このような書き込み遅延動作により、ファイルの同じ区画への冗長なI/O操作が回避されます。

ファイルを閉じるときに遅延を通知する場合、このzero-fill動作を原因とすることになります。ファイルを操作する場合、必ず以下の操作を行ってください。

- データを連続的にファイルに書き込みます。書き込みの空隙は、ファイルの保存時にゼロで埋める必要があります。
- ファイルの末尾よりも後にファイルポインタを移動しないでください。ファイルを閉じることができなくなります。
- 書き込んだデータの長さに合わせて、ファイルを切り詰めます。削除する予定のスクラッチファイルの場合、ファイルをゼロ長まで切り詰めます。

OS X Libraryディレクトリの詳細

Libraryディレクトリは、システムとコードがすべての関連データおよびリソースを保存する場所です。OS Xでは、大半がシステムにより自動的に生成される、多くの異なるサブディレクトリがこのディレクトリに格納されます。iOSでは、アプリケーションインストーラは~/Libraryに少数のサブディレクトリ（CachesやPreferencesなど）のみを作成し、アプリケーションはその他すべてのサブディレクトリの作成を受け持ちます。

表 A-1に、OS XのLibraryディレクトリで一般的に見られるサブディレクトリの一部と、内部のファイルのタイプを一覧にしています。これらのディレクトリは、必ず本来の用途で使用する必要があります。アプリケーションでもっとも多く使用されるディレクトリの詳細については、“[Libraryディレクトリはアプリケーション固有のファイルを保存](#)”（23 ページ）を参照してください。

表 A-1 Libraryディレクトリのサブディレクトリ

サブディレクトリ	ディレクトリの内容
Application Support	<p>アプリケーション固有のすべてのデータとサポートファイルを含みます。ユーザに代わってアプリケーションで作成および管理されるファイルであり、ユーザデータを含むファイルも対象です。</p> <p>慣習として、これらの項目はすべてアプリケーションのバンドルIDと名前が一致するサブディレクトリに保存される必要があります。たとえば、アプリケーション名がMyApp、バンドルIDがcom.example.MyAppであれば、アプリケーションのユーザ固有のデータファイルとリソースは、~/Library/Application Support/com.example.MyApp/ディレクトリに配置します。アプリケーションは、このディレクトリの必要に応じた作成に責任を持ちます。</p> <p>アプリケーションの実行に要求されるリソースは、アプリケーションのバンドル内に配置する必要があります。</p>
Assistants	ユーザの設定またはその他の作業を支援するプログラムを含みます。
Audio	オーディオのプラグイン、ループ、デバイスドライバを含みます。
Autosave Information	アプリケーション固有の自動保存データを含みます。

サブディレクトリ	ディレクトリの内容
Caches	<p>キャッシュされ、必要に応じて再生成できるデータを含みます。アプリケーションはキャッシュファイルの存在に依存できません。キャッシュファイルは、アプリケーションのバンドルIDと名前が一致するディレクトリに配置する必要があります。</p> <p>慣習として、アプリケーションは、アプリケーションのバンドルIDと名前が一致するディレクトリにキャッシュファイルを配置する必要があります。たとえば、アプリケーション名がMyApp、バンドルIDがcom.example.MyAppであれば、ユーザ固有のキャッシュファイルは~/Library/Caches/com.example.MyApp/ディレクトリに配置します。</p>
ColorPickers	HLS（色相、彩度、明度）ピッカーまたはRGBピッカーなど、特定のモデルに従って色を選択するリソースを含みます。
ColorSync	ColorSyncのプロファイルとスクリプトを含みます。
Components	システムのバンドルと拡張を含みます。
Containers	サンドボックス内のすべてのアプリケーションのホームディレクトリを含みます（ユーザドメインでのみ利用可能）。
Contextual Menu Items	システムレベルのコンテキストメニューを拡張するためのプラグインを含みます。
Cookies	データファイルとウェブブラウザのクッキーを含みます。
Developer	Xcodeやその他のデベロッパツールで使用するデータを含みます。
Dictionaries	スペルチェッカーの言語辞書を含みます。
Documentation	コンピュータのユーザと管理者向けのドキュメントファイルとAppleHelpパッケージを含みます（AppleHelpパッケージは、Documentation/Helpディレクトリ内にあります）。ローカルドメインでは、Appleから配布されたヘルプパッケージ（デベロッパ向けドキュメントを除く）がこのディレクトリに含まれます。
Extensions	デバイスのドライバとその他のカーネル拡張を含みます。
Favorites	頻繁にアクセスされるフォルダ、ファイル、またはウェブサイトのニックネームを含みます（ユーザドメインでのみ利用可能）。
Fonts	表示と印刷のいずれにも使用されるフォントファイルを含みます。
Frameworks	フレームワークと共有ライブラリを含みます。システムドメインのFrameworksディレクトリは、Appleで提供されるフレームワーク専用です。デベロッパは、ローカルかユーザドメインのいずれかにカスタムフレームワークをインストールする必要があります。

サブディレクトリ	ディレクトリの内容
Internet Plug-ins	ウェブブラウザのコンテンツのためのプラグイン、ライブラリ、フィルタを含みます。
Keyboards	キーボードの定義を含みます。
LaunchAgents	現在のユーザに対して起動および実行するエージェントアプリケーションを指定します。
LaunchDaemons	システム上でルートとして起動および実行されるデーモンを指定します。
Logs	コンソールと特定のシステムサービスのログファイルを含みます。ユーザはConsoleアプリケーションを使用して、これらのログを閲覧できます。
Mail	ユーザのメールボックスを含みます（ユーザドメインでのみ利用可能）。
PreferencePanes	System Preferences アプリケーションのプラグインを含みます。デベロッパはローカルドメインに、カスタム環境設定ペインをインストールする必要があります。
Preferences	ユーザの環境設定を含みます。このディレクトリ内に、デベロッパ自身がファイルを作成してはいけません。環境設定の値を取得または設定する場合、必ずNSUserDefaultsクラスまたはシステムで提供される同等のインターフェイスを使用します。
Printers	システムドメインとローカルドメインでは、このディレクトリにはプリンタの設定に必要なプリンタドライバ、PPDプラグイン、ライブラリが含まれます。ユーザドメインでは、このディレクトリにはユーザの利用可能なプリンタ設定が含まれます。
QuickLook	QuickLook プラグインを含みます。アプリケーションがカスタムドキュメントタイプを閲覧するための QuickLook プラグインを定義している場合、このディレクトリにインストールします（ユーザまたはローカルドメインのみ）。
QuickTime	QuickTime コンポーネントと拡張を含みます。
Screen Savers	スクリーンセーバーの定義を含みます。スクリーンセーバープラグインの作成に使用されるインターフェイスの詳細については、『 <i>Screen Saver Framework Reference</i> 』を参照してください。
Scripting Additions	AppleScript の機能を拡張するスクリプトおよびスクリプトリソースを含みます。

サブディレクトリ	ディレクトリの内容
Sounds	システムアラートサウンドを含みます。
StartupItems	(廃止) ブート時に実行される、システムおよびサードパーティのスク립トとプログラムを含みます (ブート時のプロセスの起動の詳細については、『 <i>Daemons and Services Programming Guide</i> 』を参照してください)。
Web Server	ウェブサーバのコンテンツを含みます。このディレクトリには、保存されるCGIスク립トとウェブページが含まれます (ローカルドメインでのみ利用可能)。

ファイルシステムの詳細

この付録には、OS XとiOSでサポートされるファイルシステムに関する情報を収めています。

サポートされるファイルシステム

OS Xは、表 B-1でリストされるものを含む、さまざまなファイルシステムとボリューム形式をサポートします。主要なボリューム形式はHFS Plusですが、OS XはUFSファイルシステムでフォーマットされたディスクからブートすることもできます。今後のOS Xのバージョンでは、ほかの形式によるブートにも対応するかも知れません。

表 B-1 OS Xでサポートされるファイルシステム

ファイルシステム	解説
HFS	Mac OS Standardファイルシステム。旧バージョンのMac OSのための標準のMacintoshファイルシステム。このタイプのファイルシステムは、OS X v10.6で読み取り専用として処理されます。
HFS Plus	Mac OS Extendedファイルシステム。OS Xのための標準Macintoshファイルシステム。
WebDAV	ウェブのファイルに直接アクセスする場合に使用されます。たとえば、iDiskはファイルへのアクセスにWebDAVを使用します。
UDF	Universal Disk Format。DVDメディア（ビデオ、ROM、RAM、RW）、および一部の書き込み可能なCD形式の標準ファイルシステム。
FAT	16および32ビット版のMS-DOSファイルシステム。FAT 12ビットはサポートされていません。
ExFAT	デジタルカメラとその他の周辺機器で使用する置き換え可能な形式。
SMB/CIFS	Microsoft Windows SMBファイルサーバとクライアントとのファイルの共有に場合に使用されます。
AFP	Apple Filing Protocol。全バージョンのMac OSのプライマリネットワークファイルシステム。

ファイルシステム	解説
NFS	Network File System。共通に使用されるUNIXファイル共有規格。OS XはTCPとUDPを使用するNFSv2とNFSv3をサポートします。OS X 10.7はNFSv4 over TCPもサポートします。
FTP	標準のインターネットFTP（File Transfer Protocol）のファイルシステムラッパー。
Xsan	ストレージエリアネットワークで使用されるAppleの64ビットクラスタファイル。
NTFS	Windowsオペレーティングシステムが動作するコンピュータの標準ファイルシステム。
CDDAFS	オーディオCDをマウントし、ディスクのオーディオトラックをAIFF-Cエンコードファイルとしてユーザに表示する場合に使用されるファイルシステム。
ISO 9660	CD-ROMで使用されるファイルシステム形式。

多くのファイルシステムは、パスの区切りにセパレータ文字の使用を要求します。文字列の構築には、手動よりもNSURLファイル構築メソッドの使用が推奨されます。

Finder

OSXのファイルシステムにアクセスするユーザは主にFinderを目的とするため、Finderのファイルの表示方式と操作方法を少しでも理解しておくに役立ちます。

ファイル名の並べ替え規則

Finderのファイルおよびディレクトリ名の並べ替え順序は、Unicodeコンソーシアムで定義されたUnicode Collation Algorithm（技術規格UTS #10）を基準としています。同規格は、すべてのUnicode文字の完全かつ明確な並べ替え順序を規定し、Unicodeコンソーシアムのウェブサイト

（<http://www.unicode.org>）で閲覧できます。Finderは、承認されたいくつかの代替動作を活用して、このアルゴリズムのデフォルトの並べ替え動作をわずかに変更します。主なものを説明します。

- 句読点と記号は並べ替えに重要です。
- 数字の部分文字列は、数の実際の文字の並べ替えとは異なり、数値に従って並べ替えられます。
- 並べ替えでは大文字と小文字は考慮されません。

ファイルとディレクトリの表示規則

Finderは複数の情報を使用して、ファイルとディレクトリの表示様式を決定します。ファイルのバンドルビット、タイプコード、クリエータコード、およびファイル名拡張子はすべて、アイコンの決定に役立ちます。ユーザ設定も役割があります。以下のステップで、ファイルとディレクトリのアイコンの選択に使用するプロセスを説明します。

- ファイル：
 1. Finderは適切なアイコンの指定をLaunch Serviceに要求します（ファイルアイコンは、通常は適切なファイルタイプを定義するアプリケーションにより指定されます。システムアプリケーションも、多くの既知のファイルタイプのデフォルトアイコンを提供します）。
 2. アイコンが利用できない場合、Finderは汎用ファイルアイコンを表示します。
- バンドルされていないディレクトリ：
 1. 特定のシステムディレクトリに対して、Finderはカスタムアイコンを表示します。これらのカスタムアイコンは、通常、汎用のフォルダアイコンに、ディレクトリの目的を示す画像が重ねられています。
 2. その他のすべてのディレクトリについては、システムは汎用フォルダアイコンを表示します。
- バンドルディレクトリ：
 1. システムはディレクトリをファイルとして表示し、デフォルトではユーザによるディレクトリ内容の表示を許可しません（この場合もユーザは、Controlキーを押しながらディレクトリをクリックし、表示されたコンテキストメニューから「パッケージの内容を表示(Show Package Contents)」を選択すると、バンドルディレクトリの内容を表示できます）。
 2. バンドルディレクトリのファイル名の拡張子が.appであれば、Finderはアプリケーションバンドルに関連したアイコンを適用します。
 3. バンドルディレクトリの場合、Finderはタイプコード、クリエータコード、ファイル名拡張子をLaunch Servicesデータベースで探し、その情報に基づいて適切なカスタムアイコンを探します。
 4. ファイルとディレクトリのいずれかに適したカスタムアイコンがない場合、Finderは所定の項目のタイプに適したデフォルトアイコンを表示します。

デフォルトのアイコンは、項目が特にドキュメント、バンドルされていないディレクトリ、アプリケーション、プラグイン、汎用バンドルのいずれであるかによって異なります。

ファイルタイプとクリエイターコード

ファイルタイプとクリエイターコードは、ファイルのタイプとファイルを作成したアプリケーションを特定する旧式の方法です。ファイルタイプコード（通常は4文字のシーケンスで指定される32ビット値）は、ファイル内の内容のタイプを特定します。クリエイターコード（同様に4文字のシーケンスを使って特定される）は、ファイルを作成したアプリケーションを特定し、そのファイルのプライマリエディタとして振る舞います。全般的にこれらのコードは廃止されていますが、従来のファイルおよびアプリケーションで、またシステム内の一部の場所では出現する場合があります。

OS Xファイルシステムのセキュリティ

OS Xは、アクセスをファイルとディレクトリ（ボリュームのマウントポイント、ハードウェアデバイスを表すブロックと文字の特殊デバイス、シンボリックリンク、名前付きパイプ、UNIXドメインソケットなどの特別なファイルとディレクトリを含む）に制限する、ファイルシステムのセキュリティポリシーを設定しています。この付録では、これらのポリシーについて、またポリシーがアプリケーションにどのように影響するかについて説明します。

セキュリティスキーム

OS Xは3つのファイルシステムセキュリティスキーム、すなわちUNIX (BSD) 権限、POSIXアクセス制御リスト (ACL)、サンドボックスエンタイトルメントを規定しています。また、BSDレイヤも、UNIX権限をオーバーライドする複数のファイル別フラグを規定しています。これらのスキームを以降のセクションで説明します。

また、OS Xでは管理ユーザは、Finderでボリュームの「情報を見る(Get Info)」を選択し、「このボリュームの所有権を無視する(Ignore ownership on this volume)」チェックボックスを選択することにより、削除可能なボリュームの所有権と権限の確認をボリューム単位で無効にすることができます。

これらの権限モデルは、以下のように組み合わせて使用されます。

1. アプリケーションのサンドボックスが要求されたアクセスを禁止する場合、要求は拒絶されます。詳細については、「[サンドボックスのエンタイトルメント](#)」（106 ページ）を参照してください。
2. 問題のボリュームに対する所有権の確認が、システム管理者により無効に設定されている場合（Finderの「情報を見る(Get Info)」のチェックボックスを使用）、要求は許可されます。
3. ファイルにアクセス制御エントリが存在する場合、アクセス権の判断のために評価および使用されます。詳細については、「[POSIXのACL](#)」（106 ページ）を参照してください。
4. ファイルフラグにより操作が禁止されている場合、操作は拒否されます。詳細については、「[BSDのファイルフラグ](#)」（113 ページ）を参照してください。
5. 上記以外の場合、ユーザIDがファイルのオーナーに一致する場合、「ユーザ」の権限（別名「オーナー」権限）が使用されます。詳細については、「[UNIX権限](#)」（114 ページ）を参照してください。

6. 上記以外の場合、グループIDがファイルのグループに一致する場合、「グループ」の権限が使用されます。詳細については、「[UNIX権限](#)」（114 ページ）を参照してください。
7. 上記以外の場合、「その他」の権限が使用されます。詳細については、「[UNIX権限](#)」（114 ページ）を参照してください。

サンドボックスのエンタイトルメント

OS Xは、サンドボックスを使用したアプリケーションのファイルへのアクセスの制御をサポートしています。これらの制限により、アプリケーションの本来の権限がオーバーライドされます。サンドボックス制限は減法的であり、加法的ではありません。したがって、アプリケーションのサンドボックスもアクセスを許可する場合、ファイルシステムの権限が、そのアプリケーションに許可される最大限のアクセスとなります。

POSIXのACL

OS X v10.4以降、MachとBSDの権限ポリシーは、権限に対する制御レベルがBSDよりもはるかに高いデータ構造である、カーネルの**ACLs**（アクセス制御リスト）のサポートにより実施されます。たとえば、システム管理者はACLを使って、個々のユーザに対してファイルの削除を許可し、ファイルへの書き込みを禁止することができます。ACLはまた、Windowsオペレーティングシステムで使用される、Active DirectoryとSMB/CIFSネットワークと互換性があります。OS Xの各種のネットワークファイルシステムにおけるACLサポートの詳細については、「[ネットワークファイルシステム](#)」（119 ページ）を参照してください。

ACLは**ACE**（アクセス制御エントリ）の順序付きリストです。各ACEはユーザまたはグループを一連の権限に関連付け、それぞれの権限の許可または拒否を指定します。ACEはまた、継承に関連した属性も含んでいます（「[権限の継承](#)」（109 ページ）を参照）。

注意 ファイルシステムのACLは、キーチェーンで使用されるACLに関連付けられていません。キーチェーンについては、『*Keychain Services Programming Guide*』で説明しています。

ファイルシステムのアクセス制御ポリシー

ファイルシステムのACLを使用すると、BSD権限のみを使用した場合よりも、詳細かつ複雑なアクセス制御ポリシーを実施することができます。この場合、BSDで使用される3つの許可よりも多い権限ビットを使用し、各権限の許可と拒否の関連性をユーザまたはグループ単位で実施します。表B-2に、ACLで使用される権限ビットを示しています。下記の権限ビットを、[表B-3](#)（115 ページ）に示すBSD権限ビットと比較してください。

表 B-2 ACLを使用したファイル権限ビット

ビット	ファイル	ディレクトリ
読み取り	読み取りファイルを開く	ディレクトリの内容を表示する
書き込み	書き込みファイルを開く	ファイルエントリをディレクトリに追加する
実行	ファイルを実行する	ディレクトリ内を検索する（またディレクトリ内のファイルまたはディレクトリにアクセスする）
削除	ファイルを削除する	ディレクトリを削除する
付加	ファイルに付加する	ディレクトリにサブディレクトリを付加する
子の削除	—	ディレクトリからファイルまたはサブディレクトリのエントリを削除する
属性の読み取り	基本属性を読み取る	基本属性を読み取る
属性の書き込み	基本属性を書き込む	基本属性を書き込む
拡張の読み取り	拡張（名前付き）属性を読み取る	拡張（名前付き）属性を読み取る
拡張の書き込み	拡張（名前付き）属性を書き込む	拡張（名前付き）属性を書き込む
権限を読み取る	ファイル権限（ACL）を読み取る	ディレクトリ権限（ACL）を読み取る
権限の書き込み	ファイル権限（ACL）を書き込む	ディレクトリ権限（ACL）を書き込む
所有権の取得	所有権を取得する	所有権を取得する

権限を変更する権利も特定の権限により制御されることに注意してください。

ACLとユーザID

OS XでACLを実装する主な理由の1つとして、SMB/CIFSなどのネットワークファイルのファイルシステムのサポートが挙げられます（“[SMB/CIFS](#)”（120 ページ）を参照）。ネットワークを通じてユーザとグループを識別するために、各ファイルまたはディレクトリはBSDで使用するローカルで一意的なUID

とGID以外に、ユニバーサルに一意なID（UUIDs）が割り当てられなければなりません。したがって、ACLが関連付けられた各ファイルまたはディレクトリは、BSDのサポートとACLのサポートにそれぞれ2つずつの、合計4つの関連IDを持ちます。

- ユーザID（UID）
- グループID（GID）
- オーナーUUID
- グループUUID

各ファイルの3つの権限（ファイルのオーナー、ファイルのグループのメンバ、その他全員に1つずつ）を指定するBSDと異なり、1つのACLを使って各ACEの各種の権限を指定できます。ACLとBSDのその他の違いとして、BSDはファイルオーナーが必ず個人になるのに対し、ACL権限スキームではユーザとグループのいずれもがファイルオーナーになることができます。ファイルがグループで所有される場合、そのGID（BSDで使用）とグループUUIDは常に首尾一貫しています（すなわち常にシンプルに、1対1でマッピングされます）。ただし、BSDはファイルのオーナーとしてのグループの概念をサポートしないため、この場合、「ユーザ以外」で所有されるものとしてファイルを特定する特殊なUIDがシステムで割り当てられ、オーナーのUUIDはグループを表します。ファイルが1人の個人で所有される場合、そのUIDとオーナーにUUIDは首尾一貫しています。

ACLを使用するファイルのオーナーは、ACLの内容に関わらず特定の取り消し不能な権限（読み取り権限と書き込み権限）を持ちます。ファイルが個人により所有される場合、グループのUUIDによりグループがファイルシステムオブジェクトに関連付けられ、特定のACEの継承に影響を及ぼしますが（[“権限の継承”](#)（109 ページ）を参照）、グループには特別な権限が発生しません。

アクセス制御リストの評価

ACLの各ACEは、一部の権限を許可または拒否します。拒否ACEは許可ACEの不在と同じ状態を意味しないことを、必ず理解してください。この場合システムは、要求されたすべての権限が許可されるか、要求された任意の権限が拒否されるまでACEを連続的に評価します。承認の要求には、（要求側エンティティを特定する）クレデンシャル（証明書）と、操作に要求される権限が含まれます。OS X v10.4以降は、以下のアルゴリズムを使用して権限を評価します（また、継承される権限の解説については、[“権限の継承”](#)（109 ページ）を参照）。

1. 要求された権限によりオブジェクトが変更され、ファイルシステムが読み取り専用になるかオブジェクトに不変がマーキングされる場合、その操作は拒否されます。
2. 要求側のエンティティがルートユーザである場合、操作は許可されます。
3. 要求側のエンティティがオブジェクトのオーナーである場合、要求者には読み取り権と書き込み権のアクセスが付与されます。これで要求が十分に満たされる場合、操作は許可されます。

4. オブジェクトがACLを使用する場合、ACLのACEは順番にスキャンされます（拒否の関連するACEは通常は、許可に関連するACEの前に配置されます）。各ACEは、要求された権限が拒否されるか、要求される権限がすべての許可されるか、またはACLの最後に達するまで、以下の基準に沿って評価されます。
 - a. ACEは適用性がチェックされます。ACEが要求されたいずれの権限も参照していない場合、そのACEは適用不可と見なされます。また、要求側エンティティはACEで指名されるエンティティと同一であるか、要求者がACEで指名されるグループのメンバでなければなりません（グループはネスト化が可能であり、外部ディレクトリサービスを使用してグループのメンバーシップを解決できます）。適用不可のACEは無視されます。
 - b. ACEが要求された権限のいずれかを拒否する場合、要求は拒否されます（ACEによる許可と拒否に関わらず、読み取り権と書き込み権がオブジェクトのオーナーに付与されることに注意してください）。
 - c. ACEが要求された権限のいずれかを許可する場合、許可される権限のリストにこの権限が追加されます。付与される権限に要求されたすべての権限が含まれる場合、要求は許可され、プロセスは停止します。リストが不完全な場合、次のACEのチェックに移行します。
5. ACEの最後に達しても、要求されたすべての権限が見つからず、オブジェクトにBSD権限も割り当てられている場合、BSDの権限に照らして満たされていない権限が確認されます。この作業により要求されたすべての権限が付与される場合、要求は許可されます。要求された権限にBSD権限（「所有権の取得」など）が割り当てられていない場合、未決と見なされ、要求は拒否されます。
6. ファイルシステムオブジェクトがACLを使用しない場合、“UNIX権限”（114 ページ）と“BSDのファイルフラグ”（113 ページ）で説明するBSDのセキュリティポリシーに従って権限が評価されます。

要求側エンティティの証明書は、ファイルを開くまたは実行しようとしているプログラムの有効なUID（すなわちEUID）と同等のものです。EUIDは通常はプロセスを実行するユーザまたはプロセスのUIDと同じですが、「オーナーまたはルートのセキュリティポリシー」 in *Security Overview* で説明するように、（setuidビットが関連する）特別な状況では異なる場合があります。

権限の継承

BSDの権限は、ファイル単位にのみ割り当てられるため、ディレクトリに割り当てられた権限は、そのディレクトリで作成された新しいファイルまたはディレクトリの権限を反映しません。ディレクトリの権限は取り囲まれた項目に適用できますが、これは1度限りの操作です。新規に作成されたファイルまたはサブディレクトリには影響しません。これらは作成時にデフォルトの権限が割り当てられます。

これに対してACLを使用する場合、新規に作成されたファイルとサブディレクトリは、取り囲むディレクトリから権限を検証できます。ディレクトリの各ACEは、以下の継承フラグを任意に組み合わせる含めることができます。

- Inherited（このACEは継承されている）。
- File Inherit（このACEはディレクトリ内で作成されたファイルから継承する必要がある）。
- Directory Inherit（このACEはこのディレクトリ内で作成されたディレクトリから継承する必要がある）。
- Inherit Only（このACEは承認時のチェックが不要）。
- No Propagate Inherit（このACEは直接の子からのみ検証する必要がある。すなわちACEは、継承時にDirectory InheritまたはFile Inheritビットを失う）。

カーネルは新しいファイルを作成すると、親ディレクトリのアクセス制御リスト全体をスキャンし、ファイル継承がマーキングされたすべてのACEをファイルのACLにコピーします。同様に、カーネルは新しいサブディレクトリを作成すると、ディレクトリ継承がマーキングされたすべてのACEをサブディレクトリのACLにコピーします。

ファイルがディレクトリにコピー&ペーストされる場合、カーネルはコピー先の新しいファイル内にソースファイルの内容を複製します。システムで新しいファイルが作成されるため、親ディレクトリのACLがチェックされ、オリジナルファイル内のすべてのACEに対して継承されたACEが追加されます。一方、ファイルがディレクトリから削除される場合、オリジナルファイルは複製されず、ACEは継承されません。この場合、移動したファイルに親ディレクトリのACEが追加されるのは、管理者が特別に、親ディレクトリからディレクトリ内のファイルとサブディレクトリにACEを伝達する場合に限られます。同様に、いったんファイルが作成されると、管理者が特別に変更を伝達しない限り、親ディレクトリのACLの変更はディレクトリ内のファイルとサブディレクトリのACLに影響しません。

BSDでは、ディレクトリの権限を囲まれるファイルとサブディレクトリに適用した場合、囲まれたオブジェクトの権限が完全に置き換えられます。これに対してACLの場合、継承されたACEはファイルまたはディレクトリ上の既存のほかのACEに追加されます。

ACLにACEが配置される順序、および権限の可否が評価される順序は以下のとおりです。

1. 明示的に指定された拒否の関連性
2. 明示的に指定された許可の関連性
3. 継承された関連性。親に現れる順序と同じ

したがって、明示的に指定されたACEは継承されたすべてのACEよりも優先されます。ACEの評価方法の詳細については、“[アクセス制御リストの評価](#)”（108 ページ）を参照してください。

ACEは継承可能であるため、管理者はディレクトリに継承可能なACEを割り当てることにより、ディレクトリで作成されたファイルの細部にわたる権限を制御できます。これにより、各ファイルに個別にACEを割り当てる作業を省くことができます。さらに、ACEはユーザグループに適用できるため、管理者はグループに権限を割り当てることができ、個人に対して権限を指定する作業は不要になります。

す。ファイルと個人ではなく、ディレクトリとグループにアクセスのセキュリティを適用することにより、多くの場合、管理者は時間を節約することができ、ファイルシステムのパフォーマンスを向上させることができます。









アプリケーションのプログラマは、権限のディレクトリからの自動的な継承は、アプリケーションが新しいファイルにACLを作成する必要性、あるいは継承されたACEをファイルが保存されるときに維持する必要性が消滅する意味であることに注意しなければなりません。これは継承されたACEを使用するファイルに、カーネルがACLを作成するためです（BSD権限の割り当てと継承は、ACLの影響を受けません。ACLがサポートされない場合、BSDの権限が使用されます。ACLとBSDの両権限が設定されているときの権限の評価方法については、“[セキュリティスキーム](#)”（105 ページ）を参照してください）。

OS X Server v10.4では、サーバ管理者は以下の操作を実行できます。

- 親ディレクトリの権限を、階層の下位のすべてのファイルとディレクトリにコピーできます。これにより、ディレクトリツリーの権限が均一化されるため、BSD権限についてのみ使用します。
- 親ディレクトリの権限を、階層の下位のすべてのファイルとディレクトリに伝達できます。この場合、明示的に指定されたACEは変更されず、ここから継承されたACEも変更されません。ファイルとサブディレクトリは、図B-1に示すように、明示的にACEを指定したディレクトリ内の元の場所に新規に作成される場合と同様にACEを継承します。
- 親ディレクトリからの継承を、特定のディレクトリまたはファイルに適用できます。
- ディレクトリ内の継承されたACEを明示化できます。
- ディレクトリとファイルからすべてのACEを削除できます。
- ボリューム上のACLを有効または無効にできます。

サーバのGUIは、ファイルのACEを直接操作できません。Finderには、ACEを設定または変更するGUIがありません。ACEは、コマンドラインツール`ls`と`chmod`を使って、サーバとクライアントの両方で読み取りおよび設定できます。

図 B-1 権限の伝達

Name	Permissions
 Directory 0	A
 File 0	a
 Subdirectory 1	a
 File 1	a
 Subdirectory 2	a
 Subdirectory 3	a
 Subdirectory 4	a
 File 2	a









ACE A = read/write for admin group,
inheritable by all children

ACE B = full control for admin group,
inheritable by all children









ACE C = full control for writers group,
inheritable by all children

Lowercase = inherited ACE









1. Server admin applies ACE C to subdirectory 2

Name	Permissions
 Directory 0	A
 File 0	a
 Subdirectory 1	a
 File 1	a
 Subdirectory 2	a, C
 Subdirectory 3	a
 Subdirectory 4	a
 File 2	a









2. Server admin propagates ACEs from subdirectory 2

Name	Permissions
 Directory 0	A
 File 0	a
 Subdirectory 1	a
 File 1	a
 Subdirectory 2	a, C
 Subdirectory 3	a, c
 Subdirectory 4	a, c
 File 2	a, c

3. Server admin removes ACE A from directory 0, adds B

Name	Permissions
 Directory 0	B
 File 0	a
 Subdirectory 1	a
 File 1	a, C
 Subdirectory 2	a, c
 Subdirectory 3	a, c
 Subdirectory 4	a, c
 File 2	a, c

4. Server admin propagates ACEs from directory 0

Name	Permissions
 Directory 0	B
 File 0	b
 Subdirectory 1	b
 File 1	b
 Subdirectory 2	b, C
 Subdirectory 3	b, c
 Subdirectory 4	b, c
 File 2	b, c

BSDのファイルフラグ

標準のUNIXファイル権限以外に、OS Xは`chflags` APIで提供される複数のBSDファイルフラグと、対応する`chflags` コマンドをサポートします。これらのフラグにより、UNIXの権限がオーバーライドされます。

Cフラグ名 コマンドライン名	16進値	意味
UF_NODUMP nodump	0x1	UNIX <code>dump</code> コマンドを使用する場合、ファイルをバックアップしません。このフラグはOS Xでは全般的に使用されません。 このフラグは、ファイルのオーナーかスーパーユーザ（root）に変更できます。
UF_IMMUTABLE uchg、uchange、またはuimmutable	0x2	ファイルの移動、名前変更、削除が行えません（シングルユーザモードのrootによる操作を除きます）。 このフラグは、ファイルのオーナーかスーパーユーザ（root）に変更できます。
UF_APPEND uappnd または uappend	0x4	ソフトウェアからファイルに不可できますが、既存のデータを修正できません。 このフラグは、ファイルのオーナーかスーパーユーザ（root）に変更できます。
UF_OPAQUE opaque	0x8	ユニオンマウントについてはディレクトリは不透過です。すなわち、基本のファイルシステムにディレクトリが存在し、同じ名前であれば、そのディレクトリの内容は表示されません。 このフラグは、ファイルのオーナーかスーパーユーザ（root）に変更できます。
-----	0x10	予約。
UF_COMPRESSED (同等のコマンドなし)	0x20	ファイルがファイルシステムレベルで圧縮されます。 このフラグは、ファイルのオーナーかスーパーユーザ（root）に変更できます。
-----	0x40-0x4000	予約。
UF_HIDDEN hidden	0x8000	GUIでファイルを非表示にすることを示唆します。 このフラグは、ファイルのオーナーかスーパーユーザ（root）に変更できます。

Cフラグ名 コマンドライン名	16進値	意味
SF_ARCHIVED arch または archived	0x10000	ファイルがアーカイブされています。 このフラグはスーパーユーザ（root）のみが変更できます。
SF_IMMUTABLE schg、schange、ま たはsimmutable	0x20000	ファイルの移動、名前変更、削除が行えません（シングルユーザモードのrootによる操作を除きます）。 このフラグはスーパーユーザ（root）のみが変更できます。
SF_APPEND sappnd または sappend	0x40000	ソフトウェアからファイルに不可ですが、既存のデータを修正できません。 このフラグはスーパーユーザ（root）のみが変更できます。

注意 chflagsを使ってフラグを無効にする場合、適宜、フラグ名の前にnoを追加するか、先頭のnoをドロップします。

UNIX権限

各ファイルシステムオブジェクトは、3つの属性で一連のUNIX権限が定義されています。

- **UID**、User IDの短縮形。通常はファイルのオーナーを示します。
- **GID**、グループIDの短縮形。
- **フラグ**、権限ビットとその他の関連する属性を含みます。

ファイルまたはディレクトリのフラグは、多くの場合は3文字または4文字の8進値で表わされる16ビット値です（先頭4ビットまたは7ビットはドロップ）。

- ビット12～15：ファイルのタイプを示すフラグ。これらのビットは不変であり、権限を表示するときに省略されます。
- ビット9～11：表B-4（115ページ）で説明する特別な権限ビット。通常は0です。設定されていない場合は省略できます。
- ビット6～8：Owner権利ビット。これらのビットは、ファイルまたはディレクトリのUIDに等しい有効なユーザID（EUID）を持つプロセスにアクセスを制限します。

これらのビットの優先順位は最高です。

- ビット3～5：Group権利ビット。これらのビットは、ファイルまたはディレクトリのGIDに一致する有効なグループID（EGID）を持つプロセスにアクセスを制限します。

これらの権利は、ファイルまたはディレクトリのUIDに一致するEUIDを持つプロセスには適用されません。これらのビットの優先順位はOwner権利よりも下位ですが、Other権利よりも上位です。

- ビット0～2：Other権利ビット。これらのビットは、ファイルまたはディレクトリのUIDとGIDのいずれにも一致しないプロセスに適用されます。

Owner、Group、Otherビットセットは、読み取り、書き込み、実行（短縮形：rwx）の3ビットを含みます。これらのビットの影響は、表 B-3に示すように、ファイルとディレクトリでは異なります。

表 B-3 BSDのファイル権限ビット

ビット	ファイル	ディレクトリ
読み取り	読み取りファイルを開くことができる	ディレクトリの内容を表示できる
書き込み	書き込みファイルを開くことができる	ディレクトリの内容を変更できる（囲まれたファイルまたはディレクトリの移動、名前変更、または削除）
実行	ファイルを実行プログラムとして処理できる	ディレクトリ内を検索できる（ディレクトリ内のファイルまたはディレクトリにアクセスする）

r、w、xの各ビット以外に、ファイルシステムオブジェクトごとに3つの補助権限ビット、setuid、setgid、stickyが割り当てられます。

表 B-4 ファイルシステムの特別な権限ビット

ビット	ファイル	ディレクトリ
setuid	バイナリ実行ファイルの場合、実行されると、最終プロセスのEUIDは親プロセスのEUIDではなく、ファイルのUIDに設定されます。 最終プロセスのRUIDは、通常どおり親プロセスのEUIDに設定されます。 このフラグは、解釈されたスクリプトまたは非実行ファイルには影響しません。	ディレクトリ内で作成されたファイルまたはディレクトリのUIDは、ディレクトリのUIDに設定されません。

ビット	ファイル	ディレクトリ
setgid	バイナリ実行ファイルの場合、実行されると、最終プロセスのEGIDは親プロセスのEGIDではなく、ファイルのGIDに設定されます。 最終プロセスのRGIDは、通常どおり親プロセスのEGIDに設定されます。 このフラグは、解釈されたスクリプトまたは非実行ファイルには影響しません。	ディレクトリ内で作成されたファイルまたはディレクトリのGIDは、ディレクトリのGIDに設定されます。
sticky	OS Xでは影響がありません。 ポータビリティ上の理由により、このビットの設定は避けてください。ほかのバージョンのUNIXに影響するためです。	囲まれたファイルまたはディレクトリの削除を、以下の3ユーザに制限します。 <ul style="list-style-type: none">ファイルまたはディレクトリのオーナー（EUID = ファイルUID）root（EUID = 0）stickyディレクトリのオーナー（EUID = ディレクトリUID）

たとえば、バイナリ実行ファイルのオーナーがルートユーザであり、**setuid**ビットが設定されている場合、プログラムは常にEUID=0で実行されます。このようなプログラムは、root以外の誰かにより実行された場合にルート権限で実行されるため、セキュリティ上の脆弱性が発生する可能性があります。したがって、**setuid**と**setgid**プログラム作成と使用を制限することが重要です。

ユーザは自分の保有するファイルに対してのみ、権限を変更できます。したがって、rootが所有するプログラムで**setuid**ビットを設定できるのはルートユーザのみです。

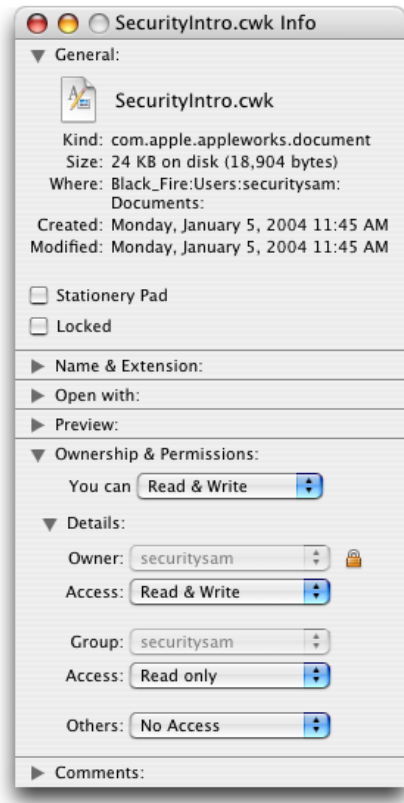
UNIXの権限は、ターミナルとFinderでユーザに表示されます。ターミナルでは、以下のように表示されます。

```
$ ls -ld filename dirname
drwxr-xr-x  2 username  groupname  68 Jun 16 13:40 dirname
-rw-r--r--  1 username  groupname   0 Jun 16 13:40 filename
```

権限の形式（左）の詳細は、『*ShellScriptingPrimer*』の「シェルスクリプトのセキュリティ」で説明しています。

Finderでは、UNIX権限は、ファイルまたはフォルダのInfoダイアログに所有権と権限の情報の形式で表示されます（図 B-2）。

図 B-2 所有権と権限の情報



特別なユーザとグループ

このセクションでは、昇格された特権を持つ、または昇格された権限を取得するための権利を持つOS Xの個々のユーザとグループについて説明します。

ルートユーザ

ルートユーザは、主要なシステムプロセスの多くを所有し、コンピュータに接続するデバイスのファイルシステムオブジェクトに無制限にアクセスできます。ルートユーザが可能な操作の例を以下に挙げます。

- 任意のファイルの読み取り、書き込み、実行
- 任意のファイルまたはフォルダのコピー、移動、名前変更
- 任意のファイルの所有権の移転と権限の再設定

標準のBSD権限セマンティクスとOS X実装の主な違いは、OS Xではシステムのインストール後にルートユーザが無効になっていることです。ほとんどの場合、管理者はrootで実行する必要がありません（“[管理者グループ](#)”（119ページ）を参照）。sudoユーティリティを使用してルート権限を担うことも可能です。sudoユーティリティは、ルートユーザの有効化を要求しませんが、その起動はターミナルアプリケーションに限定されます。すなわち同ユーティリティを使用するマシンに物理的にアクセスする必要があります。用途の詳細については、sudoのmanページを参照してください。

ルートユーザは、ユーザのシステム上で有効にすることはできません。アプリケーションがルートユーザとして操作を実行することを要求する場合、承認サービスを使用しなければなりません。詳細については、セキュリティドキュメントの『*Authorization Services C Reference*』および『*Authorization Services Programming Guide*』を参照してください。

注意 ほとんどの場合、ルートユーザを有効化せずに、adminグループのメンバとして実行するかsudoを使用します。ルートユーザの有効化が絶対的に必要になる場合、NetInfo Managerユーティリティを実行し、自らをローカル管理者として承認します。次に「セキュリティ (Security)」メニューから「ルートユーザを有効にする (Enable Root User)」を選択します。このメニュー項目は、ローカルのadminグループ、すなわち特別な管理特権を持つグループのメンバであり、すでにローカルドメインで承認されている場合にのみ有効に使用できます。ルートユーザを有効化すると、パスワードは空白に示されるため、「セキュリティ (Security)」メニューから「ルートパスワードを変更する (Change Root Password)」を選択して、ルートユーザにパスワードを割り当てる必要があります。ルートアクセスに必要な作業が終了したら、「セキュリティ (Security)」メニューから「ルートユーザを無効にする (Disable Root User)」を選択して、ルートユーザの特権を放棄する必要があります。

ほとんどのユーザ権限はネットワーク全体に適用されるのに対し、setuidとsetgidはルートユーザの概念により、多くの場合ネットワークボリューム上で無視されます。

たとえば、NFSからリモートボリュームにアクセスする場合、デフォルトではルートユーザはnobody、すなわち、ほとんどアクセスを持たない特別なユーザにマップされます。これにより、特定のコンピュータ上のルートユーザが、別のコンピュータのルートユーザになることが回避されます。

ホイールグループ

BSDにはホイールグループと呼ばれる特別なグループがあります。ホイールグループのメンバーシップは、コマンドラインでsuユーティリティを使用してルートユーザになることが可能なユーザに発生します。ホイールグループに所属しないユーザは、正しいパスワードを保有している場合でも、ルートユーザになることができません。

OS X v 10.3以降では、ホイールグループは使用されません。その機能はadminグループにより継承されました。

管理者グループ

OS Xでは、ルートユーザの代わりにadminグループを使用します。**管理者グループ**のメンバ（**管理者**と称される）は、ルートユーザが実行できるほぼすべての機能を実行でき、Finderを使用して実行できます。すなわちコマンドラインに頼る必要がありません。管理者が禁止されている操作は、システムドメインでのファイルの直接の追加、修正、削除です。ただし、管理者はインストーラやソフトウェア更新などの特別なアプリケーションを使用して、この操作を実行できます。

システムにOS Xをインストールしたユーザは、自動的にシステムの最初の管理者になります。その後、このユーザ（またはその他の管理者）は、「アカウント(Accounts)」環境設定を使用して新規ユーザのアカウントをローカルシステムで作成し、システムの任意のユーザに管理特権を付与することができます。

ネットワークファイルシステム

このセクションでは、ネットワークファイルサーバのプロトコルによる権限の使用について説明します。OS Xは、4つのネットワークファイルサーバプロトコルをサポートします。

- **AFP**—Apple Filing Protocol。Mac OS 9システムの主たるファイル共有プロトコルです。AppleShareサーバとクライアントにより使用されます。
- **NFS**—Network File System。UNIXシステムで使用される主要なファイル共有プロトコル。
- **SMB/CIFS**—Server Message Block/Common Internet File System。WindowsとUNIXシステムで使用されるファイル共有プロトコル。
- **WebDAV**—Web-based Distributed Authoring and Versioning。ウェブ上でコラボレイティブなファイル管理を可能にするHTTPの拡張版。

AFP

AppleShareクライアントとサーバがいずれもAFP 3.0をサポートする場合、実際のBSD権限は接続経由で送信されます。AFPサーバ上のファイルまたはディレクトリがACLを使用する場合、ACLは接続経由で送信され、有効な権限がFinderにより表示されます。ただし、権限の行使はサーバ上でのみ行われ、クライアントでは行われません。OS XのACLの実装の詳細については、“[POSIXのACL](#)”（106 ページ）を参照してください。

接続でAFP 2.xを使用している場合、権限の機能の違いに注目してください。

- BSDはファイルの権限をサポートするのに対し、AFP 2.xはサポートしません。
- BSDは「ベストマッチ」の権限ポリシーを実施します。オーナーであれば、オーナー権限を取得します。オーナーではなく、ファイルグループに所属している場合は、グループ権限を取得します。それ以外は、その他の権限を取得します。AFPは累積的な権限ポリシーを実施します。オー

ナー、グループ、その他の権限から導出された権限の集合体が権限になります。たとえば、フォルダがグループによる書き込みが可能であり、オーナーによる書き込みが禁止されている場合、AFPの権限ではオーナーはフォルダを修正できますが、BSDの権限では修正できません。

- BSDはフォルダの`rw`ビットを、表 B-3に示すように解釈します。AFPの権限では、同ビットを「ファイル参照」、「フォルダ参照」、「変更を実行」として定義しています。AppleShare 2.x サーバを扱う場合、OS X AppleShareクライアントはこれらの特権モデル間でマッピングを行います。同様のマッピングは、AppleShare 2.xクライアントを使用してOS Xサーバに接続する場合にも適用されます。
- ACLはAFP 2.xではサポートされません。

AFPは、ネットワーク経由のデータへのアクセスから、EUIDが0のプロセス（すなわち`root`として実行されているプロセス）を除外します。

NFS

一般に、NFSはセキュアなプロトコルではありません。ほとんどのNFSサーバがクライアントを信頼しているためです。すなわち、このファイル操作がユーザBobに代わって行われたとクライアントが主張すれば、サーバはユーザBobの代わりに操作を実行します。ただし、クライアント側でルートアクセスが可能である場合、ユーザBobに成り済まし、NFSサーバ上のBobのすべてのファイルにアクセスできます。一定のセキュリティを維持するために、ほとんどのNFSサーバはルートユーザを、ファイルもディレクトリも所有しない特別なユーザ`nobody`にマップします。このため、EUIDが0であれば、一般にNFSサーバでは「その他」へのアクセスを許可するファイルにのみアクセスできます。

SMB/CIFS

SMBは、Windowsネットワークで共通に使用されるファイル共有のためのネットワークプロトコルです。CIFSは多くの場合、SMBと同じ概念で使用されます。**Samba**は、UNIX上でSMB/CIFSサーバを実行するソフトウェアです。したがって、このファイル共有プロトコルは、SMB、CIFS、SMB/CIFS、Samba、Windowsファイル共有などさまざまに呼ばれます。

OS X v10.4以降では、SMB/CIFSに対応したアクセス制御リスト（ACL）が実装されています。個々のユーザはACLの設定や変更が行えませんが、サーバ管理者は実行できます（管理者は、SMBサーバのコマンドラインを使用してACLを操作しますが、クライアントとサーバがいずれも同じActive Directory ドメインにバインドされている場合に限られます）。ただし、権限の行使はサーバ上でのみ行われ、クライアントでは行われません。OS XのACLの実装の詳細については、“[POSIXのACL](#)”（106 ページ）を参照してください。

OS X v10.3以前では、OS XのすべてのSMBアクセス制御はサーバ上で実行され、クライアントでは実行されません。そのため、OS XユーザがSMBファイルサーバをマウントする場合、マウントされたボリューム、ディレクトリ、またはファイルは、読み取り、書き込み、実行のアクセスが可能、また

ユーザによる所有が可能とFinderに表示されます。ただし、ユーザがフォルダまたはファイルを開こうとすると、サーバはユーザのアクセス権限を評価し、アクセスを許可するか、アクセスを許可する前に新規のユーザ名とパスワードの入力をユーザに要求します。

SMB/CIFSの権限の詳細、およびそれぞれの動作の修正方法の詳細については、SMBのmanページ（man 5 smb.conf）を参照してください。

WebDAV

WebDAVプロトコルは、リモートからの、すなわちネットワーク接続によるウェブコンテンツの書き込みと編集をユーザに許可するHTTPプロトコルです。OS XのWebDAVファイルシステムは、WebDAV要求とHTTP要求を使用して、WebDAVに対応したHTTPサーバ上のリソースにファイルとディレクトリとしてアクセスします。

WebDAVプロトコルは、ユーザとグループをサポートしません。さらに、WebDAVクライアントは、WebDAVサーバ上のファイルとディレクトリへのアクセスを試みる前に、それらのアクセス権限を判断できません。したがって、OS XのWebDAVファイルシステムは、すべてのファイルとディレクトリに対してユーザとグループのIDをunknownに設定し、権限を全員、すなわちユーザ、グループ、その他のread、write、executeにデフォルト設定します。

WebDAVファイルシステムがWebDAVに対応したHTTPサーバに要求を送信すると、サーバは承認が必要かどうかを判断します。承認が不要であれば、サーバは要求を受け付けます。承認が必要であれば、サーバは認証の資格情報（ユーザ名とパスワードなど）をチェックし、それらが存在し正しい場合、サーバはクライアントを承認し、アクセスを許可します。承認が必要であり、資格情報が送信されない、または資格情報が正しくない場合、サーバは認証のチャレンジにより要求を拒否します。ユーザが正しい資格情報を供給できない場合、WebDAVファイルシステムはアクセスを拒否します。

WebDAVファイルシステムで使用されるプロトコルの詳細については、以下のドキュメントを参照してください。

- *Hypertext Transfer Protocol-HTTP/1.1* <http://www.ietf.org/rfc/rfc2616.txt>
- *HTTP Authentication: Basic and Digest Access Authentication* <http://www.ietf.org/rfc/rfc2617.txt>
- *HTTP Extensions for Distributed Authoring-WebDAV* <http://www.ietf.org/rfc/rfc2518.txt>

iOSのファイルシステムのセキュリティ

iOSの基本の権限モデルはOSXと同じですが、実際にはiOSアプリケーションはサンドボックスに制限され、全般的にアプリケーションで作成されたファイルにのみアクセスできます（アプリケーションは、その他のアドレス帳のデータや写真などの特定のファイルにアクセスできますが、その用途に特別に指定されたAPIを使用した場合に限定されます）。

また、iOSデバイス上でファイル保護が有効に設定されている場合、アプリケーションはデバイスがロックされているときの特定のファイルへのアクセスを禁止することができます。これは、ユーザの個人データや機密情報を含むファイルに対して行います。

キーチェーンと同様に、iOSの暗号化されたファイルは、モバイル機器のバックアップ内でも暗号化されます。暗号の有効化以外に、バックアップ時のファイルの完全な非表示も指定できます。

Important ファイル保護をサポートするアプリケーションは、保護されたファイルが使用不可のときにアプリケーションが実行されるという状況に対処できなければなりません。

ファイル保護APIに関する詳細については、『*iOS App Programming Guide*』の“Advanced App Tricks”を参照してください。

書類の改訂履歴

この表は「ファイルシステムプログラミングガイド」の改訂履歴です。

日付	メモ
2012-03-08	iOSアプリケーションが新規ディレクトリを生成できる場所に関する記述を修正しました。
2012-03-01	OS XアプリケーションにApp Sandboxを採用したとき、ファイルシステム機能の動作が変わる旨の注意を、概要の項に追加しました。
2012-01-09	<p>ファイルコーディネータとファイルプレゼンタに関する情報を追加しました。</p> <p>iCloudの情報のコピー、移動、削除を追加しました。</p> <p>『セキュリティの概要』で説明していた権限情報を追加し、また各種の権限モデルの相互作用を示す概要を追加しました。</p>
2011-06-06	<p>ファイルシステムのファイル、ディレクトリ、その他の内容を作成および管理する方法を記述した新規ドキュメント。</p> <p>このドキュメントは、『ファイルシステムの概要』、『下位ファイル管理のプログラムのトピック』、『アプリケーションのファイルの管理』で説明していた情報を扱っています。</p>



Apple Inc.
© 2012 Apple Inc.
All rights reserved.

本書の一部あるいは全部を Apple Inc. から書面による事前の許諾を得ることなく複写複製（コピー）することを禁じます。また、製品に付属のソフトウェアは同梱のソフトウェア使用許諾契約書に記載の条件のもとでお使いください。書類を個人で使用する場合に限り1台のコンピュータに保管すること、またその書類にアップルの著作権表示が含まれる限り、個人的な利用を目的に書類を複製することを認めます。

Apple ロゴは、米国その他の国で登録された Apple Inc. の商標です。

キーボードから入力可能な Apple ロゴについても、これを Apple Inc. からの書面による事前の許諾なしに商業的な目的で使用すると、連邦および州の商標法および不正競争防止法違反となる場合があります。

本書に記載されているテクノロジーに関しては、明示または黙示を問わず、使用を許諾しません。本書に記載されているテクノロジーに関するすべての知的財産権は、Apple Inc. が保有しています。本書は、Apple ブランドのコンピュータ用のアプリケーション開発に使用を限定します。

本書には正確な情報を記載するように努めました。ただし、誤植や制作上の誤記がないことを保証するものではありません。

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/>

App Store is a service mark of Apple Inc.

iCloud is a registered service mark of Apple Inc.

iDisk is a registered service mark of Apple Inc.

Apple, the Apple logo, AppleScript, AppleShare, Carbon, Cocoa, ColorSync, Finder, Instruments, iPhoto, iTunes, Keychain, Keynote, Mac, Mac OS, Macintosh, Numbers, Objective-C, OS X, Pages, QuickTime, Sand, Xcode, and Xsan are trademarks of Apple Inc., registered in the United States and other countries.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

UNIX is a registered trademark of The Open Group.

Apple Inc. は本書の内容を確認しておりますが、本書に関して、明示的であるか黙示的であるかを問わず、その品質、正確さ、市場性、または特定の目的に対する適合性に関して何らかの保証または表明を

行うものではありません。その結果、本書は「現状有姿のまま」提供され、本書の品質または正確さに関連して発生するすべての損害は、購入者であるお客様が負うものとしします。

いかなる場合も、Apple Inc. は、本書の内容に含まれる瑕疵または不正確さによって生じる直接的、間接的、特殊的、偶発的、または結果的損害に対する賠償請求には一切応じません。そのような損害の可能性があらかじめ指摘されている場合においても同様です。

上記の損害に対する保証および救済は、口頭や書面によるか、または明示的や黙示的であるかを問わず、唯一のものであり、その他一切の保証にかわるものです。Apple Inc. の販売店、代理店、または従業員には、この保証に関する規定に何らかの変更、拡張、または追加を加える権限は与えられていません。

一部の国や地域では、黙示あるいは偶発的または結果的損害に対する賠償の免責または制限が認められていないため、上記の制限や免責がお客様に適用されない場合があります。この保証はお客様に特定の法的権利を与え、地域によってはその他の権利がお客様に与えられる場合もあります。