Core Dataプログラミングガイド

Cocoa > Design Guidelines

2011-08-03

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

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

目次

管理オブジェクト 39

管理オブジェクトのアクセサメソッド 45

管理オブジェクトの生成と削除 55

管理オブジェクトのフェッチ 61

管理オブジェクトの使い方　65

Core Dataによるメモリ管理　75

関係とフェッチ済みプロパティ　79

非標準の永続属性型　89

管理オブジェクトの検証　99

フォールティングと一意化　105

永続ストアの使い方　111

Core DataとCocoaバインディング　115

変更管理　119

図、表、リスト

フォールティングと一意化　105

変更管理　119

Core Dataプログラミングガイドを読む前に

Core Dataフレームワークは、オブジェクトの永続性をはじめ、ライフサイクルやオブジェクトグラフの管理に関連して、頻繁に必要となるさまざまな処理機能を、汎用化、自動化して提供します。

対象読者

この文書ではCore Dataフレームワークの使い方を解説します。*Cocoa開発の基本事項、特にプログラミング言語Objective-Cやメモリ管理については、よく知っている読者を想定します。*

> **重要：** Core Dataフレームワークの基本事項をひと通り解説しますが、この技術の使い方を効率よく学ぶ上で、先頭から順に読んでいく、という方法は向いていません。「参考文献」に載っているチュートリアルも併用して理解を補うようお勧めします。推奨する学習手順については、『*Core Data Starting Point*』を参照してください。

この書類の構成

以下の各章では、Core Dataフレームワークが解決しようとしている課題と実際の解決策、基本的な機能、このフレームワークがよく利用される作業について説明します。

- 「技術の概要」（17 ページ）では、Core Dataとは何か、このフレームワークを利用するとどのような利点があるのか、を概説します。
- 「Core Dataの基本事項」（21 ページ）では、この技術の基本的なアーキテクチャを解説します。
- 「管理オブジェクトモデル」（29 ページ）では、管理オブジェクトモデルの特徴を説明します。
- 「管理オブジェクトモデルの使い方」（33 ページ）では、実際のアプリケーションの開発に管理オブジェクトモデルを取り入れる方法を示します。
- 「管理オブジェクト」（39 ページ）では、管理オブジェクトを実装した`NSManagedObject`クラスの機能と、カスタムクラスを実装してエンティティを表す手順およびその利点を解説します。
- 「管理オブジェクトのアクセサメソッド」（45 ページ）では、カスタム管理オブジェクトにアクセサメソッドを実装する方法を説明します。
- 「管理オブジェクトの生成と削除」（55 ページ）では、管理オブジェクトのインスタンスを生成、削除する、プログラムの記述方法を示します。

- 「管理オブジェクトのフェッチ」（61 ページ）では、管理オブジェクトをフェッチする手順と、これを効率よく行うための検討事項を解説します。
- 「管理オブジェクトの使い方」（65 ページ）では、実際のアプリケーションで管理オブジェクトを操作する際の考慮事項を説明します。
- 「Core Dataによるメモリ管理」（75 ページ）では、Core Data環境におけるメモリ管理について解説します。
- 「関係とフェッチ済みプロパティ」（79 ページ）では、管理オブジェクト間の関係やそのモデル化の手順を示し、操作に当たっての考慮事項を説明します。また、フェッチ済みプロパティについても述べます。これは弱い片方向の関係のようなものです。
- 「非標準の永続属性型」（89 ページ）では、色、Cの構造体など、標準以外の型の属性を用いる方法を示します。
- 「管理オブジェクトの検証」（99 ページ）では、検証の種類、検証メソッドの実装方法を説明し、どの時点で検証するべきか、について解説します。
- 「フォールティングと一意化」（105 ページ）では、オブジェクトグラフの大きさを抑制する方法、管理オブジェクトコンテキストの範囲で管理オブジェクトの重複がないことを保証する方法を解説します。
- 「永続ストアの使い方」（111 ページ）では、永続ストアの作成方法、ある型のストアを別の型に移行する手順、ストアのメタデータを管理する方法について説明します。
- 「Core DataとCocoaバインディング」（115 ページ）では、Core DataがどのようにCocoaバインディングを統合し、その能力を活かしているか、を示します。
- 「変更管理」（119 ページ）では、管理オブジェクトコンテキストや永続スタックが複数ある場合に起こりうる、さまざまな問題について解説します。
- 「永続ストアの特徴」（125 ページ）では、各種のストアの特徴を述べ、特にSQLiteストアの動作設定について説明します。
- 「Core Dataによる並列処理」（131 ページ）では、Core Dataアプリケーションで並列プログラミングを行う方法を解説します。
- 「Core Dataの処理性能」（135 ページ）では、Core Dataアプリケーションの処理性能を改善するためのさまざまな技法を示します。
- 「Core Dataのトラブルシューティング」（143 ページ）では、Core Dataを使った開発でよく発生する問題とその解決方法を説明します。
- 「効率的なデータのインポート」（153 ページ）では、データをCore Dataアプリケーションにインポートする手順を解説します。
- 「Core Dataに関してよく訊ねられる事項」（159 ページ）では、Core Dataに関してよく訊ねられる質問に答えます。
- 「用語解説」（171 ページ）では、Core Dataでよく使われる用語の意味を解説します。

関連項目

次の資料も参照してください。

- *Core Data Starting Point*
- *Core Data Tutorial for iOS*
- *Core Data Utility Tutorial*
- *Core Data Model Editor Help*
- *Core Data Snippets*
- *ManagedObjectDataFormatter*（Xcode用プラグイン）

技術の概要

この章では、Core Dataの基本機能を挙げ、この技術を導入するとどのような利点があるのか、を解説します。

Core Dataの機能

Core Dataフレームワークは、オブジェクトの永続性をはじめ、ライフサイクルやオブジェクトグラフの管理に関連して、頻繁に必要となるさまざまな処理機能を、汎用化、自動化して提供します。次のような機能があります。

- 変更の追跡、取り消し処理の支援。

  取り消し/再実行の処理を支援する機能が組み込まれています。単なるテキスト編集の取り消し/再実行にとどまらない、包括的な機構です。

- 関係の管理。

  変更の影響範囲（オブジェクト間の関係の一貫性を含む）を管理できます。

- フォールティング。

  プログラムのメモリ消費量を削減できるよう、オブジェクトの遅延読み込み機構が組み込まれています。また、部分的な実体化や、「コピーオンライト」方式によるデータ共有にも対応しています。

- プロパティ値の自動検証。

  Core Dataの管理オブジェクトは、標準的な「キー値コーディング」検証メソッドを拡張し、個々の値が許容範囲内であること、複数の値の組み合わせが妥当であることを検証できるようになっています。

- スキーマの移行。

  アプリケーションのスキーマ変更に対処するのは、開発の手間、実行時資源の両面で難しいことがあります。Core Dataのスキーマ移行ツールを使えばその作業を削減でき、場合によってはその場で効率的に移行できてしまうこともあります。

- アプリケーションのコントローラ階層との最適な統合による、ユーザインターフェイス同期の支援機能。

  iOS上で動作する`NSFetchedResultsController`オブジェクトを利用して、Mac OS XのCocoaバインディングと統合できます。

- 完全自動化されたキー値コーディングおよびキー値監視。

キー値コーディングおよびキー値監視に対応した、属性のアクセサメソッドを自動生成する機能に加え、対多関係を対象として、コレクションの形でやり取りするアクセサも生成できます。

- データのグループ化、しぼり込み、組織化（メモリ中、ユーザインターフェイス上の両方）。
- 外部データリポジトリにデータを格納する処理の自動化。
- 洗練されたクエリのコンパイル。

  SQLを記述する代わりに、`NSPredicate`オブジェクトにフェッチ要求を組み込むことにより、複雑なクエリを生成できます。`NSPredicate`は、基本的な問い合わせ機能だけでなく、相関サブクエリその他、高度なSQLの操作機能も備えています。さらに、正式なUnicodeやロケールを意識した、検索、整列、正規表現などの機能も組み込まれています。

- ポリシーのマージ。

  版の追跡機能、楽観的ロック機構が組み込まれており、複数のプロセスによる書き込みの衝突を自動的に解消できます。

Core Dataの導入による利点

Core Dataにはさまざまな利点があります。分かりやすい利点としては、アプリケーションのモデル階層を実装するために必要なコード量（行数）が、一般に50～70%程度削減できる、ということがあります。これは主として、先に挙げた機能による恩恵です。Core Dataに組み込まれている機能は、改めて実装しなくてよいからです。さらに、別途テストする必要も、最適化について意識する必要もありません。

Core Dataには念入りに実装されたコードが組み込まれています。その品質はユニットテストにより確保され、多くの顧客がさまざまなアプリケーションに組み込んで日常的に使っているのです。フレームワークは版を重ねるごとに最適化の度合いを増しています。普通であれば、アプリケーションレベルのコードではまず利用されることのないような、モデルや実行時ルーチンの情報を駆使しています。さらに、セキュリティ管理やエラー処理においても洗練された機能が提供されるのに加え、競合する製品に比べても、最も優れたメモリ拡張性を備えています。問題領域に合わせて最適化したシステムを、長い時間をかけて独自に構築したとしても、Core Dataを使って容易に達成できる性能を越えることはできないでしょう。

フレームワーク自体の長所に加え、Core Dataには、Mac OS Xのツール群に適切に統合されている、という利点があります。モデル設計ツールを使えば、グラフィック画面上で迅速かつ容易にスキーマを作成できます。Instrumentsアプリケーションのテンプレートを使って、Core Dataの性能を測定したり、さまざまな問題点をデバッグしたりすることも可能です。Mac OS Xのデスクトップに目を向けると、Core DataはInterface Builderも統合しているので、モデルからユーザインターフェイスを生成できます。このような特性が、アプリケーションの設計、実装、デバッグのサイクルをさらに短縮するために役立っているのです。

Core Dataは「何でないか」

Core Dataとは何か、どんなことができるのか、なぜ有用なのか、という観点から見てきましたが、よくある誤解を避けるためにも、逆に「何でないか」を説明しておくことにしましょう。

- Core Dataは関係データベースでも関係データベース管理システム（RDBMS）でもありません。

  Core Dataは、変更管理や、ストレージにオブジェクトを保存し、検索するための基盤を提供します。永続ストアの一種としてSQLiteを使うことは可能ですが、これ自身がデータベースというわけではありません（これは、たとえばメモリ内ストアを思い浮かべればよく理解できるでしょう。変更追跡や変更管理には使えますが、実際にファイルにデータを保存することはありません）。

- Core Dataはいわゆる「銀の弾丸」ではありません。

  コードを記述しなくてもよい、というわけではないのです。Xcodeのデータモデリングツールと Interface Builderだけを使って、洗練されたアプリケーションを作成することもできなくはありませんが、現実的なアプリケーションを構築するためには、コードを記述する必要があります。

- Core DataはCocoaバインディングに依存しません。

  Core DataにはCocoaバインディングが適切に統合され、同等の技術を活用しています。また、両者の相乗効果により、記述するべきコード量を大幅に削減できます。しかし、Cocoaバインディングなしでも、Core Dataを使うことは可能です。ユーザインターフェイスを備えていないCore Dataアプリケーションも作成できます（『*Core Data Utility Tutorial*』を参照）。

Core Dataの基本事項

この章では、Core Dataの基本的なアーキテクチャと、フレームワークの使い方を解説します。

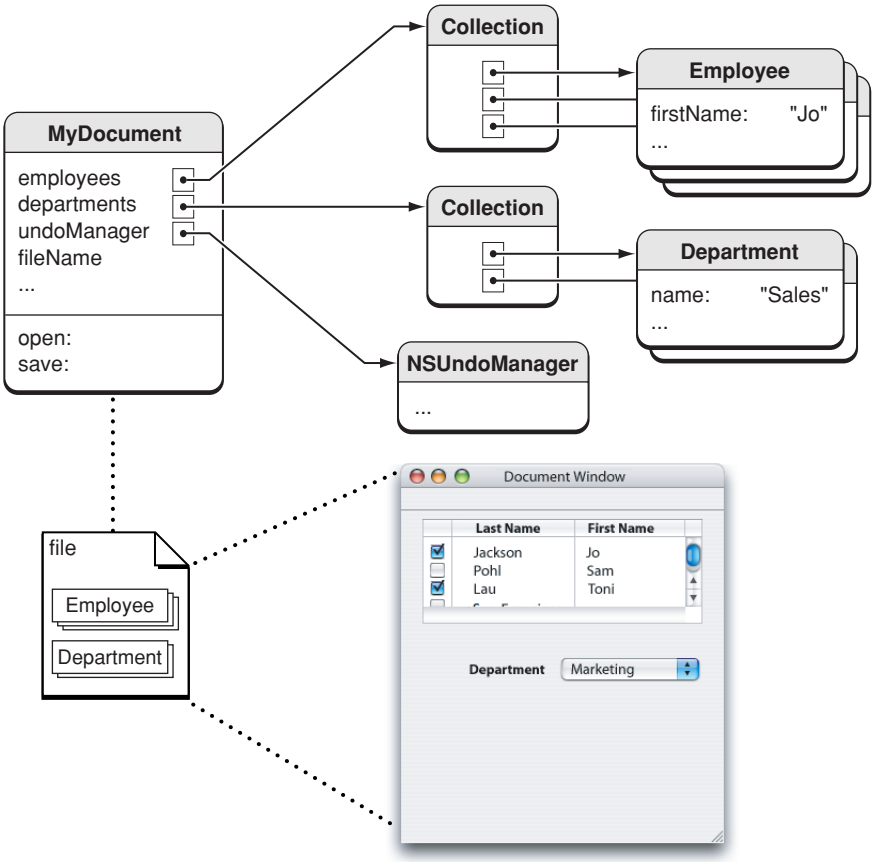
Core Dataの基本的なアーキテクチャ

大部分のアプリケーションで、オブジェクトのアーカイブを格納したファイルを開き、少なくとも1つ存在するルートオブジェクトを参照する手段が必要になります。さらに、オブジェクトをすべてファイルにアーカイブ保存する機能、（取り消し機能を組み込む場合は）オブジェクトに対して施した変更を追跡する機能も必要です。たとえば従業員管理アプリケーションの場合、「Employee（従業員）」オブジェクトや「Department（部署）」オブジェクトがアーカイブ保存されたファイルを開く手段、「すべてのEmployee（従業員）の配列」のようなルートオブジェクトを参照する手段がなければなりません（図 1 （22 ページ）を参照）。これに加えて、すべての「Employee」、すべての「Department」をファイルにアーカイブ保存する手段も必要でしょう。

注 ： この資料では、便利さや分かりやすさを考慮して、従業員管理の例で説明します。対処するべきさまざまな問題が潜んでおり、その内容が理解しやすい、という点で好都合なのです。しかし、Core Dataフレームワークの用途は、データベース型のアプリケーションに限定されているわけでも、クライアントサーバ型の動作を想定しているわけでもありません。Sketchのようなベクトルグラフィック用アプリケーションの基盤としても、Keynoteのようなプレゼンテーション用アプリケーションの基盤としても、フレームワークは等しく有用です。

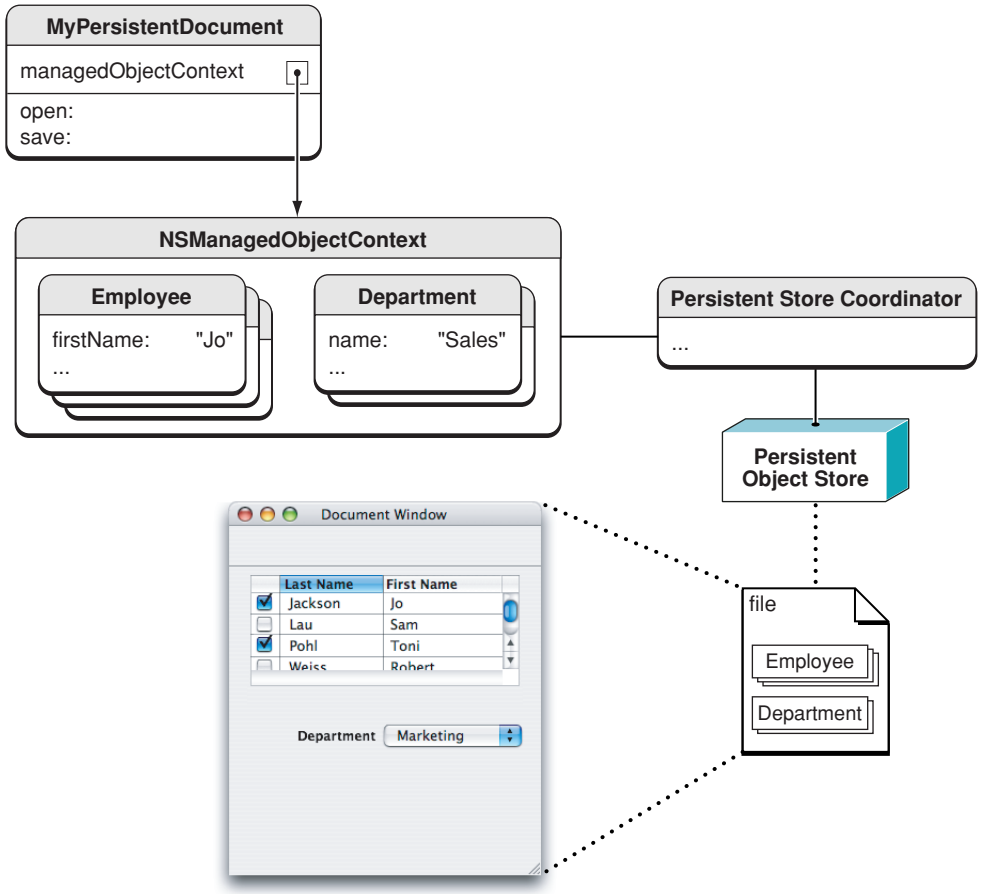
こういったタスクを管理するコードを、すべてではないにせよ、記述する必要があります。たとえばMac OS Xのデスクトップ上では、Cocoa文書アーキテクチャが提供するアプリケーション構造や機能を利用してかなりの作業を削減できますが、それでも若干のメソッドは記述しなければなりません。データをアーカイブ保存し、逆にそれを読み込むメソッド、モデルオブジェクトを追跡するメソッド、取り消しマネージャとやり取りして取り消し処理を実現するためのメソッドなどです。

図 1　標準的なCocoa文書アーキテクチャによる文書管理



Core Dataフレームワークを導入すれば、それだけでこういった機能の多くが使えるようになります。主として**管理オブジェクトコンテキスト**（あるいは単に「コンテキスト」）と呼ばれるオブジェクトを用います。管理オブジェクトコンテキストには、基盤となるフレームワークオブジェクトのコレクション（**永続スタック**）に対するゲートウェイとして、アプリケーション上のオブジェクトと、外部データストレージとの間を仲介する働きがあります。スタックの一番底にあるのは**永続オブジェクトストア**です（図 2 （23 ページ）を参照）。

図 2　Core Dataによる文書管理



Core Dataを活用できるのは、文書ベースのアプリケーションに限りません。ユーザインターフェイスを持たない、Core Dataベースのユーティリティを作成することも可能です（『*Core Data Utility Tutorial*』を参照）。さまざまな種類のアプリケーションを、同じ考え方で開発できるのです。

管理オブジェクトとコンテキスト

管理オブジェクトコンテキストとは、インテリジェントな「作業領域」であると考えると分かりやすいでしょう。オブジェクトを永続ストアからフェッチする場合、実際には一時的に作業領域にコピーし、オブジェクトグラフ（あるいはそのコレクション）を構築します。このオブジェクトに対して必要な修正を行いますが、変更を保存しない限り、永続ストア自体は元のままです。

モデルオブジェクトのうち、Core Dataフレームワークに関連付けられたものを、**管理オブジェクト**と呼びます。管理オブジェクトはすべて、管理オブジェクトコンテキストに登録しなければなりません。オブジェクトをグラフに追加する操作も、グラフから削除する操作も、コンテキストを介して行います。コンテキストは、個々のオブジェクトの属性に対して施された変更、オブジェクト間の関係に対して施された変更の、両方を追跡しています。これは、いつでも変更を取り消し、あるいは再実行できるようにするためです。さらに、オブジェクト間の関係に対して変更を施しても、オブジェクトグラフの整合性は確実に保たれます。

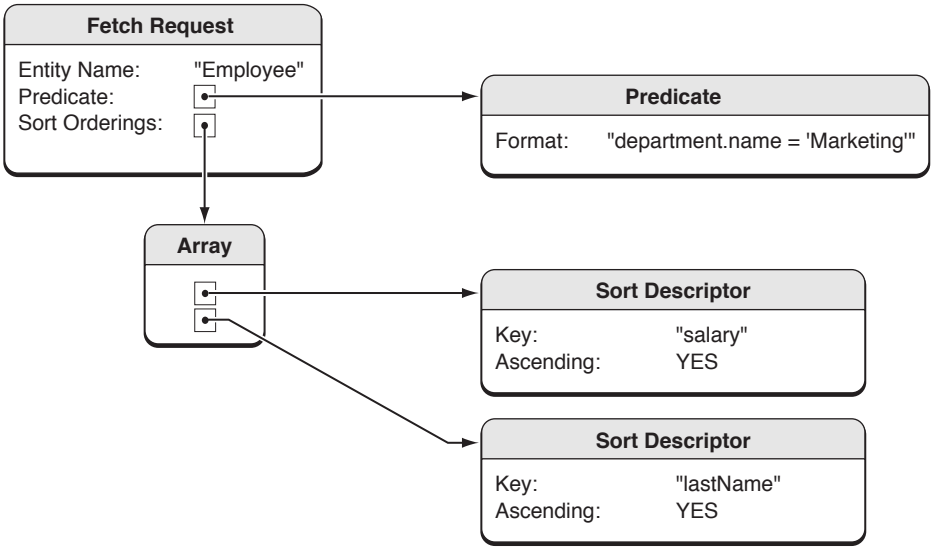
変更結果を保存する際、オブジェクトが「妥当な」状態かどうか検証するのもコンテキストの役割です。妥当な状態である場合に限り、変更内容を永続ストアに書き込みます。すなわち、新たに生成したオブジェクトに対応するレコードを追加し、削除したオブジェクトに対応するレコードを削除するのです。

管理オブジェクトコンテキストは、ひとつのアプリケーションに複数あっても構いません。永続ストアの各オブジェクトに対応して、それぞれのコンテキストには、最大でも1個の管理オブジェクトが存在します（詳しくは「フォールティングと一意化」（105 ページ）を参照）。逆に言うと、永続ストアの同じオブジェクトを、複数のコンテキストで同時に編集できます。各コンテキストには、永続ストアのオブジェクトに対応する管理オブジェクトがそれぞれあって、独立に編集できるのです。そのため、保存の際に不整合が生じる可能性があります。Core Dataにはこれに対処する手段がいくつかあります（たとえば「管理オブジェクトの使い方」（65 ページ）を参照）。

フェッチ要求

管理オブジェクトコンテキストを使ってデータを検索するためには、**フェッチ要求**を生成する必要があります。フェッチ要求とは、どのようなデータが必要か、を指定するオブジェクトのことです。「すべての従業員」、「マーケティング部門の全従業員を、給与額の順序で」などといった形で指定します。フェッチ要求は3つの部分から成ります。最低限、エンティティ名は指定しなければなりません（したがって同時にフェッチできるのは1種類のエンティティのみ）。これに加えて、「述語オブジェクト」としてオブジェクトが合致しなければならない条件を指定し、「整列記述子オブジェクト」の配列の形でフェッチしたオブジェクトの並び順を指定することもできます（図 3（24 ページ）を参照）。

図 3　フェッチ要求の例



フェッチ要求を管理オブジェクトコンテキストに送ると、指定に合致するオブジェクト群が、永続ストアに関連付けられたデータ源から返されます（該当するオブジェクトが見つからない場合もあります）。管理オブジェクトはすべて、管理オブジェクトコンテキストに登録することになっているので、フェッチ結果として得られるオブジェクトは自動的に、フェッチ処理に用いたコンテキストに登録されます。しかし先に説明したように、永続ストアの各オブジェクトに対応して、最大でも1個の管理オブジェクトが、コンテキストごとに存在します（「フォールティングと一意

化」（105 ページ）を参照）。フェッチしたオブジェクトに対応する管理オブジェクトが、はじめから該当するコンテキストに登録されていれば、（改めて登録することなく）その管理オブジェクトをフェッチ結果として返すようになっています。

フレームワークは、できるだけ効率よく処理しようと試みます。Core Dataは要求駆動方式で実装されているので、実際に必要になっていないオブジェクトを、前もって生成しなくても構いません。オブジェクトグラフが永続ストア内のオブジェクトすべてを反映しているとは限らないのです。永続ストアを指定するだけでは、データオブジェクトは管理オブジェクトコンテキストに取り込まれません。オブジェクト群を永続ストアからフェッチする操作をした時点で、要求に合致するオブジェクトだけがコンテキストに取り込まれます。オブジェクトの関係をたどって参照しようとしたオブジェクトが、まだフェッチしていないものであれば、その時点で自動的にフェッチされます。オブジェクトの使用をやめれば、自動的に解放されます（もちろん、オブジェクトグラフから削除するわけではありません）。

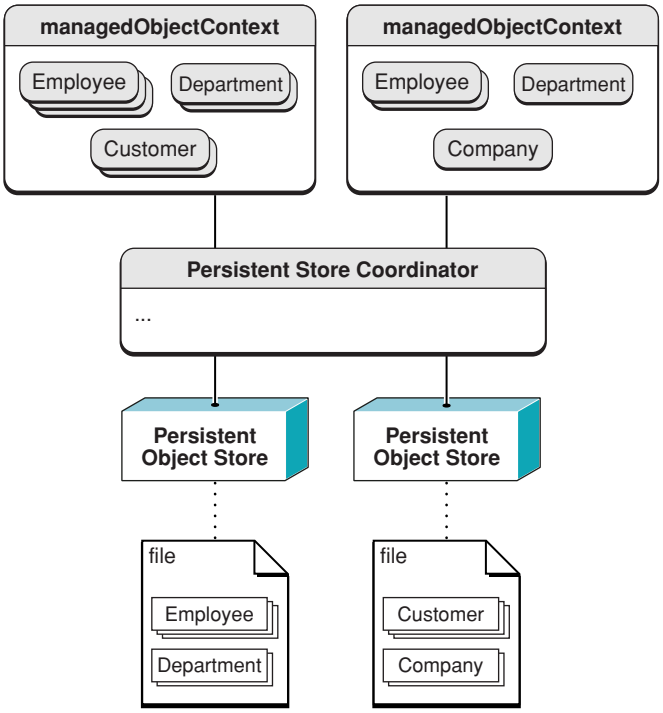
永続ストアコーディネータ

先に説明したように、アプリケーション上のオブジェクトと外部データストアとの間を仲介する、フレームワークオブジェクトのコレクションのことを、一括して永続スタックと呼びます。スタックの最上位にあるのは管理オブジェクトコンテキスト、最下位にあるのは永続オブジェクトストアです。管理オブジェクトコンテキストと永続オブジェクトストアの間には、**永続ストアコーディネータ**があります。

永続ストアコーディネータはスタックの実質的な本体です。外部からは管理オブジェクトコンテキストのように見えるよう設計されているので、永続ストアのグループも、集約された単一のストアのように見えます。したがって管理オブジェクトコンテキストは、コーディネータの支配下にあるデータストアすべてを反映する形で、オブジェクトグラフを生成できます。コーディネータが関連付けられる管理オブジェクトモデルは1つに限ります。いくつかのエンティティを別々のストアに分けて格納するためには、管理オブジェクトモデルに設定を定義することにより、モデルエンティティを分割する必要があります（「設定」（32 ページ）を参照）。

例として、図 4（26 ページ）に、Employee（従業員）とDepartment（部署）をあるファイルに、Customer（顧客）とCompany（会社）を別のファイルに格納している様子を示します。オブジェクトをフェッチ/保存しようとすれば、自動的に適切なファイルを対象として検索/アーカイブするようになっています。

図 4 高度な永続スタック



永続ストア

永続オブジェクトストアには、単一のファイルその他の外部データストアが関連付けられており、当該ストアのデータと、管理オブジェクトコンテキスト内のオブジェクトとを、対応づけて管理する働きがあります。通常、永続オブジェクトストアとのやり取りが必要になるのは、（ユーザが文書を開く、あるいは保存するなど）アプリケーションに関連付ける外部データストアの場所を新たに指定するときだけです。ほかにもCore Dataフレームワークとのやり取りは発生しますが、多くの場合、管理オブジェクトコンテキストを介して行います。

アプリケーションコードは（特に管理オブジェクトと関連付けられたアプリケーションロジックにおいて）、データを格納する永続ストアに関して、どのような想定もするべきではありません。Core Dataは何種類かの形式のファイルを扱えるようになっており、アプリケーションの用途に応じて選択可能です。ある段階でファイル形式を変更したとしても、アプリケーションアーキテクチャが変わることはありません。さらに、アプリケーションが適切に抽象化されていれば、将来フレームワークが増強されたとき、何の作業もせずにその利点を享受できます。たとえば、当初の実装では、ローカルファイルシステムからしかレコードをフェッチできないかもしれません。しかし、データの取得元に関して何の仮定もせずに実装していれば、将来、新しい種類のリモート永続ストアに対応したとき、コードはまったくそのままで、この新しい永続ストアを使えるようになるはずです。

重要： Core Dataは永続ストアとしてSQLiteも使えるようになっていますが、SQLiteのデータベースであれば何でも管理できるわけではありません。SQLiteデータベースを使うためには、データベース自身をCore Dataが生成、管理するようになっていなければならないのです。ストアの種類については、「永続ストアの特徴」（125 ページ）を参照してください。

永続文書

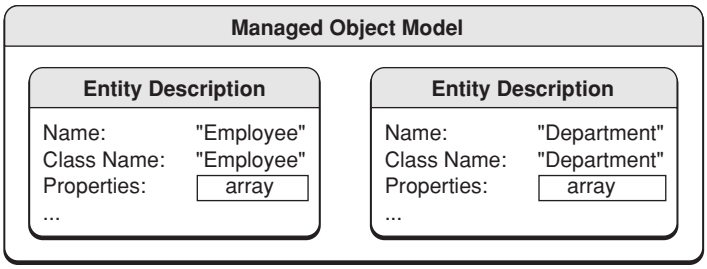
永続スタックはプログラムで生成、設定することも可能です。しかし、作成したいアプリケーションの多くは、文書ベースの、ファイルを読み込んで永続スタックを組み立てる形のものでしょう。`NSPersistentDocument`は`NSDocument`のサブクラスで、Core Dataフレームワークの機能を容易に活用できるように設計されています。`NSPersistentDocument`のインスタンスは、何もしなくても、管理オブジェクトコンテキストと永続オブジェクトストアひとつから成る永続スタックを、自動的に生成するようになっています。この場合、文書と外部データストアの間には、1対1の対応関係があることになります。

`NSPersistentDocument`クラスには、文書の管理オブジェクトコンテキストにアクセスするメソッドがあるほか、Core Dataフレームワークを介してファイルを読み書きする、`NSDocument`の標準的なメソッドも呼び出せます。オブジェクトの永続性を管理するために、別途コードを記述する必要はありません。永続文書の「取り消し」機能は、管理オブジェクトコンテキストに組み込まれています。

管理オブジェクトと管理オブジェクトモデル

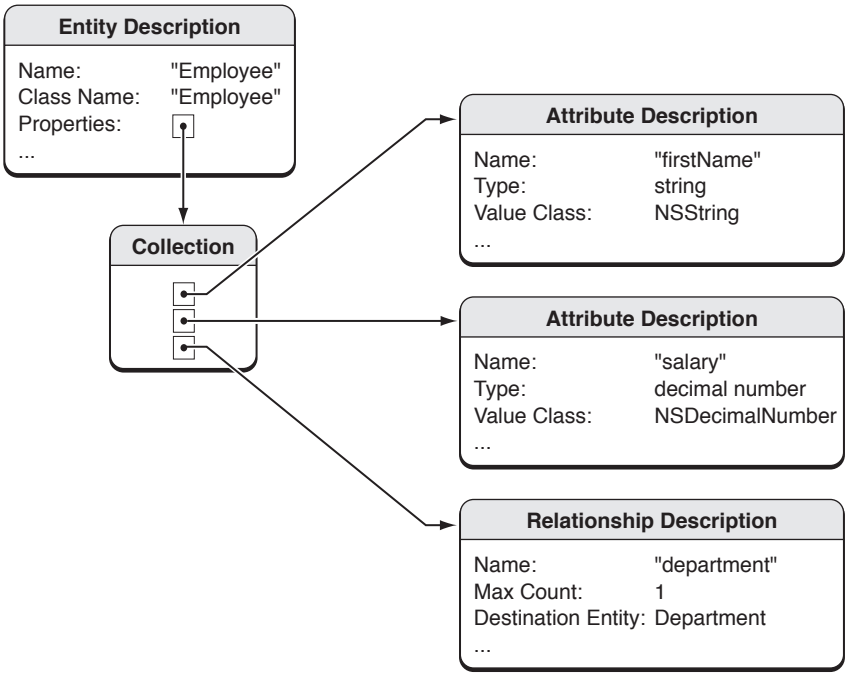
オブジェクトグラフを管理するため、およびオブジェクトの永続性を確保するために、Core Dataでは、処理対象オブジェクトに関するさまざまな記述が必要です。**管理オブジェクトモデル**とは、アプリケーションが使う、管理オブジェクトあるいはエンティティに関する記述に用いるスキーマです（図 5（27 ページ）を参照）。通常、管理オブジェクトモデルは、Xcodeのデータモデル設計ツールのグラフィック画面上で作成します（実行時にプログラムでモデルを構築することも可能）。

図 5 2つのエンティティから成る管理オブジェクトモデル



モデルは、エンティティに関するメタデータを記述した、エンティティ記述オブジェクトのコレクションから成ります。メタデータとしては、エンティティの名前、アプリケーションでこのエンティティを表すクラスの名前（エンティティ名と同じでなくても可）、属性および関係があります。そして、属性と関係はそれぞれ、属性記述オブジェクト、関係記述オブジェクトで表されます（図 6（28 ページ）を参照）。

図 6　2つの属性と1つの関係を持つエンティティ記述



管理オブジェクトは、`NSManagedObject`またはそのサブクラスのインスタンスでなければなりません。`NSManagedObject`はどのようなエンティティでも表せます。プライベートな内部ストアを使ってプロパティを管理するほか、管理オブジェクトであるために必要な基本機能をすべて実装しています。管理オブジェクトは、自分自身をインスタンスとして生成する元になったエンティティの、エンティティ記述の参照を保持しています。このエンティティ記述から、自分自身に関するメタデータ、すなわち、対応するエンティティの名前、その属性や関係に関する情報などを参照できます。また、`NSManagedObject`のサブクラスを定義し、追加機能を実装することも可能です。

管理オブジェクトモデル

Core Dataの機能の多くはスキーマに依存しています。したがって、スキーマを作成し、アプリケーションのエンティティやそのプロパティ、およびエンティティ間の関係を記述する必要があります。スキーマは管理オブジェクトモデルで表現しますが、これは`NSManagedObjectModel`のインスタンスです。一般に、モデルの記述が詳しいほど、Core Dataのきめ細かな支援機能を利用できるようになります。この章では、管理オブジェクトモデルの機能、その作成方法、アプリケーションでの使い方を説明します。

管理オブジェクトモデルの機能

管理オブジェクトモデルは`NSManagedObjectModel`のインスタンスです。ここにスキーマ、すなわちエンティティのコレクションを記述し、アプリケーションで利用します（「エンティティ」や、これに関連する「プロパティ」、「属性」、「関係」などの用語については、まず「Core Dataの基本事項」（21 ページ）、次に「Cocoa Design Patterns」の「オブジェクトのモデリング」を参照してください）。

エンティティ

モデルには、そのエンティティを表す`NSEntityDescription`オブジェクトがあります。エンティティには重要な設定項目が2つあります。名前と、実行時にエンティティを表現するクラスの名前です。エンティティそのものと、エンティティの表現に用いるクラス、エンティティのインスタンスである管理オブジェクトは、それぞれ別の概念なので、きちんと区別しなければなりません。

`NSEntityDescription`オブジェクトは、`NSAttributeDescription`および`NSRelationshipDescription`というオブジェクトを持ち、いずれもスキーマにおけるエンティティのプロパティを表します。エンティティは「フェッチ済みプロパティ」を持つことがあり、これは`NSFetchedPropertyDescription`のインスタンスで表されます。また、モデルは「フェッチ要求」テンプレートを持つことがあり、これは`NSFetchRequest`のインスタンスで表されます。

モデルでは、エンティティは継承階層上に位置づけられます。抽象エンティティとすることも可能です。

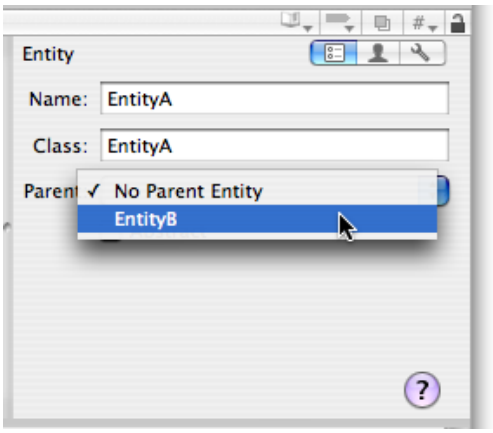
エンティティの継承

エンティティの継承は、クラスの継承と同じように考えればよいでしょう。これが役立つ状況も同様です。よく似たエンティティが多数ある場合、共通のプロパティを抽出し、スーパーエンティティとしてまとめることができるのです。多数のエンティティに同じプロパティを繰り返し指定する代わりに、1回だけ定義し、それをサブエンティティに継承させればよいことになります。たとえば、「Person」というエンティティを定義し、属性として「firstName」と「lastName」を設定すれば、そのサブエンティティとして定義した「Employee」や「Customer」に、この属性が継承されます。

多くの場合、エンティティに対応するカスタムクラスも実装し、サブエンティティに対応するクラスに継承させることになるでしょう。複数のエンティティに共通のビジネスロジックは、個別に実装するのではなく、1箇所にのみ記述してサブクラスに継承させるのです。

Xcodeのデータモデリングツールでモデルを作成する場合、エンティティの親は、「エンティティ情報」ペインの「親」ポップアップメニューで指定します（図 1 （30 ページ）を参照）。

図 1　Xcodeで親エンティティを選択している様子



一方、コード上でエンティティの継承階層を生成する場合は、階層の上位から下に向かって順に構築していく必要があります。あるエンティティのスーパーエンティティを直接設定することはできません。できるのは、（親の側に）サブエンティティを設定することだけです（`setSubentities:` メソッド）。したがって、スーパーエンティティを設定するためには、該当するスーパーエンティティの側に、自分自身を要素として含む、サブエンティティの配列を設定する必要があります。

抽象エンティティ

エンティティが「抽象」である、すなわち、そのインスタンスは生成しない、という指定ができます。通常、抽象エンティティが有用なのは、ある共通の（親）エンティティから継承する（特殊化する）形でいくつものエンティティを定義したいけれども、親エンティティ自身のインスタンスを生成することはない、という状況です。たとえば描画アプリケーションで、「Graphic」エンティティを定義し、X/Y座標、色、境界などの属性を定義することが考えられます。「Graphic」そのもののインスタンスを生成することはありません。具象サブエンティティとして「Circle」、「TextArea」、「Line」などを定義し、そのインスタンスを生成することになるでしょう。

プロパティ

エンティティのプロパティには、「属性」と「関係」の2種類があります。「フェッチ済みプロパティ」も含みます。ほかの特性と同じように、プロパティにも名前と型があります。属性にはデフォルト値を設定することもできます。プロパティ名は、`NSObject`または`NSManagedObject`の、引数なしのメソッド名と同じであってはなりません。したがって、たとえば「description」というプロパティ名は使えないことになります（`NSPropertyDescription`を参照）。

「一時プロパティ」とは、モデルの一部として定義するプロパティのうち、エンティティインスタンスのデータの一部として、永続ストアに格納することがないものを言います。Core Dataは一時プロパティに施された変更も追跡するので、その取り消し操作も可能です。

注： 一時プロパティに施した変更を取り消す場合、それがモデル化されていない情報を用いるものであれば、Core Dataは、変更前の値を渡して設定アクセサを呼び出す、という方法をとらず、単にスナップショット情報で上書きするだけです。

属性

Core Dataには、文字列、日付、整数など、さまざまな属性型が組み込まれています（ここに例示した型はそれぞれ、`NSString`、`NSDate`、`NSNumber`のインスタンスとして表されます）。組み込み以外の属性型を使いたい場合は、「非標準の永続属性型」（89 ページ）に説明する方法を用いてください。

属性値がオプションである、すなわち、値がなくても構わない旨を指定できます。しかし一般には、このような指定はお勧めできません。特に数値型の場合、値が必須の属性を用い、モデルでデフォルト値`0`を指定する方が、通常はよい結果が得られます。その理由として、SQLは`NULL`がからんだとき、比較に関してObjective-Cの`nil`とは異なる振る舞いをする、という点があります。データベースにおいては、`NULL`と`0`は同じでないので、ある列の値が`0`であるレコードを検索したとき、`NULL`である列は見つかりません。

```
false == (NULL == 0)
false == (NULL != 0)
```

さらに、データベースにおける`NULL`は、空文字列や空のデータblob（Binary Large OBject）とも異なります。

```
false == (NULL == @"")
false == (NULL != @"")
```

「関係」については、以上のような問題はありません。

関係

Core Dataでは、対1、対多の関係と、**フェッチ済みプロパティ**を扱えるようになっています。フェッチ済みプロパティは、弱い片方向の関係を表します。従業員と部署の例では、部署の「フェッチ済みプロパティ」として、「最近雇用した従業員」が考えられます（従業員の側から出る、逆向きの関係はありません）。

関係の随意性（必須か否か）、基数（対1か対多か）、削除規則を指定できます。通常は関係を両方向のものとしてモデル化するべきです。多対多の関係とは、関係とその逆がいずれも対多であるものを言います。関係について詳しくは、「関係とフェッチ済みプロパティ」（79 ページ）を参照してください。

フェッチ要求テンプレート

`NSFetchRequest`クラスを使ってフェッチ要求を記述し、永続ストアからオブジェクトを検索できます。あらゆる場合に同じフェッチ要求を実行することもありますが、フェッチ要求の一部が可変要素になっていて、その部分をユーザが指定した上で実行する、ということも多いでしょう。たとえば、出版物をすべて検索するフェッチ要求で、何という著者が、どの日付以降に出版したものを対象とするかを、実行時にユーザが指定する、という使い方です。

フェッチ要求は、事前に定義し、管理オブジェクトモデルに「名前付きテンプレート」として格納しておくことができます。したがって、クエリを事前に定義しておき、必要に応じてモデルから検索することができるのです。通常、フェッチ要求テンプレートは、Xcodeのデータモデリングツールを使って定義します（『*Xcode Tools for Core Data*』を参照）。テンプレートには変数を含めることができます（図 2を参照）。

図 2　Xcodeの述語ビルダ

フェッチ要求テンプレートについて詳しくは、「実行時に管理オブジェクトモデルにアクセスし使用する方法」（35 ページ）を参照してください。

ユーザ情報辞書

管理オブジェクトモデルの要素（エンティティ、属性、関係）の多くは、ユーザ情報辞書を持っています。ここには、キーと値の組の形で、どのような情報でも格納できます。よく格納される情報としては、エンティティの版の詳細、フェッチ要求の述語中に埋め込まれた変数に与える値などがあります。

設定

設定は、名前と、これに結び付けられたエンティティ群から成ります。同じエンティティが複数の設定に現れていても構いません。設定の定め方には、プログラム上で`setEntities:forConfiguration:`を呼び出す方法と、Xcodeのデータモデリングツールを使う方法（『*Xcode Tools for Core Data*』を参照）があります。ある設定名のエンティティを検索するためには、`entitiesForConfiguration:`を使います。

通常、設定は、エンティティごとに、別々のストアに分けて格納したい場合に使います。永続ストアコーディネータが持てる管理オブジェクトモデルは1つに限るので、あるコーディネータに関連付けられた各ストアは、同じエンティティしか持てません。この制約を回避するため、使いたいエンティティすべての合併集合を含むモデルを作成できます。次に、使いたいエンティティの部分集合それぞれに対応して、モデル内に設定を作成します。すると、コーディネータを生成する際に、このモデルを使えるようになります。ストアを追加する際には、設定ごとに異なるストア属性を指定します。ただし、設定を作成する際には、ストア間をまたがる関係を作ることはできないことに注意してください。

管理オブジェクトモデルの使い方

この章では、実際のアプリケーションに管理オブジェクトモデルを導入する手順を解説します。

管理オブジェクトモデルの作成と読み込み

モデルの作成は、通常、Xcodeで行います（『*Core Data Model Editor Help*』を参照）。また、プログラムでモデル全体を生成することも可能です（リスト 3（38 ページ）、『*Core Data Utility Tutorial*』を参照）。もっとも、そのためにはかなりのコードを記述しなければならないので、実用的なのはごく単純なアプリケーションに限ります（この場合でも、モデリングツールが何をするものなのか、チュートリアルを読んで把握しておくとよいでしょう。特に、モデルと言ってもその実体は単にオブジェクトのコレクションである旨を、理解しておくことが大切です）。

データモデルのコンパイル

データモデルは、稼働サイトに配備して用いる資源です。Xcodeで作成するモデルには、エンティティやプロパティの詳細を記述しますが、ほかに、ツールに描画するために必要な、要素の配置や色なども記述されています。これは実行時には必要ありません。そこで、モデルコンパイラ`momc`で余分な情報を削り、実行時に少しでも効率的に読み込めるようにします。コンパイルすると、「ソース」ディレクトリ`xcdatamodeld`は稼働環境におけるディレクトリ`momd`に、「ソース」ファイル`xcdatamodel`は稼働環境に配備するファイル`mom`になります。

`momc`は`/Developer/usr/bin/`以下にあります。ビルドスクリプトを別途作成し、ここからコンパイラを起動する場合は、「`momc source destination`」という形式で記述してください。ここで*`source`*はコンパイルしようとするCore Dataモデル、*`destination`*は出力先のパスを表します。

データモデルの読み込み

実行時にモデルを読み込むためのコードは、状況によっては記述する必要がありません。文書ベースのアプリケーションをMac OS X上で使う場合、モデルを検索し、読み込む処理は、`NSPersistentDocument`が管理します。また、Core Dataを基盤とする非文書アプリケーション（Mac OS X用またはiOS用）を、Xcodeを使って開発すれば、アプリケーションデリゲートに、モデルを検索するためのコードが組み込まれます。モデル名（ディスクに保存する際のファイル名）は、何であっても実行時には関係ありません。Core Dataがモデルを読み込んでしまった後は、ファイル名には意味がなくなるので、好きなように決めて構わないのです。

自分でモデルを読み込む場合、次の2通りの方法があります。

- URLを指定して単一のモデルを読み込むためには、インスタンスメソッド`initWithContentsOfURL:`を使います。

一般にはこの方法を推奨します。通常、アプリケーションには単一のモデルしかないので、この方法で問題はありません。また、URLを指定して複数のモデルをそれぞれ読み込み、`modelByMergingModels:`でマージしてから、コーディネータのインスタンスを生成することも可能です。

モデルが複数ある（特に、同じスキーマの、版の違いに応じて複数のモデルがある）場合は、どのモデルを読み込むか、意識して操作する必要があります（実行時に、同じエンティティが含まれるモデルをひとつのコレクションにマージすると、名前が衝突してエラーになる恐れがあります）。このメソッドは、アプリケーションのバンドル外にモデルを格納しており、したがってファイルシステムのURLを指定して参照する必要がある場合にも有用です。

- バンドルのコレクションを指定して、マージしたモデルを作成することも可能です。それにはクラスメソッド`mergedModelFromBundles:`を使います。

  このメソッドは、モデルの区別がそれほど重要でない場合に有用でしょう。たとえば、アプリケーションとそれがリンクするフレームワークの、どちらのモデルを使っても結果が同じであると分かっているような状況です。このクラスメソッドで、モデルをすべて、まとめて読み込むことができます。それぞれの名前を意識することも、該当するモデルがすべて見つかるよう、特別な初期化コードを置く必要もありません。

複数のモデルがあるプロジェクトで起こりうる問題

モデルを読み込む際、問題が生じうる状況がいくつかあります。多くの場合、以前ビルドした（旧版の）製品がプロジェクト内に残っている状況で、クラスメソッド`mergedModelFromBundles:`を呼び出したために起こります。

- 単にモデルファイルの名前を変更しただけでは、Core Dataが現行版と旧版のモデルをマージしようとするため、次のようなエラーが生じます。

  ```
  reason = "'Can't merge models with two different entities named 'EntityName''";
  ```

- 旧モデルにあったのとは異なるエンティティが含まれる新しいモデルを作成した場合、Core Dataは旧モデルと新モデルをマージしようとします。ストアがすでに存在すれば、それを開こうとした時点で次のようなエラーが生じます。

  ```
  reason = "The model used to open the store is incompatible with the one used to
   create the store";
  ```

この問題は、次のいずれかの方法で解消できます。

- 以前ビルドした製品をすべて削除してから、アプリケーションを実行する方法。アプリケーションバンドル自体に旧モデルファイルが残っていることがありますが、そのアプリケーションは削除してしまって構いません

- `mergedModelFromBundles:`の代わりに`initWithContentsOfURL:`でモデルを初期化する方法。URLでモデルを一意に特定できるので、旧モデルと現行モデルがマージされることはありません。

スキーマ変更により、モデルが以前のストアとの互換性を損なう問題

モデルとは永続ストアに格納するデータの構造を記述したものなので、その一部を書き換えてスキーマを変更すると、それまでに作成したストアとの互換性が損なわれ、開けなくなってしまいます。したがって、スキーマを変更した場合は、既存のストアのデータを新しい版に移行する必要があります（『*Core Data Model Versioning and Data Migration Programming Guide*』を参照）。たとえば、新しいエンティティを追加したり、既存のエンティティに属性を追加したりすると、旧版のストアは開けなくなります。これに対し、検証制約を追加したり、属性のデフォルト値を新たに設定したりしても、開けなくなることはありません。

重要： モデルを変更し、なおかつ旧モデルで作成したストアも開けるようにしたい場合は、（当該モデルを版管理し、ひとつの版として）*旧モデルを保存しておく必要があります*。適合するモデルがないストアを開くことはできません。したがって、モデルを変更し、なおかつ既存のストアも開けるようにしたい場合は、次のように操作してください。

1. モデルが版管理されているかどうか確認し、そうでなければ、現行のモデルを版管理の対象にしてください。
2. *スキーマを編集する前に*、現行モデルの新しい版を作成します。
3. この新しい版を編集します。旧版はそのまま残しておいてください。

実行時に管理オブジェクトモデルにアクセスし使用する方法

実行時にモデルにアクセスしたい場合があります。たとえば、フェッチ要求テンプレート、ローカライズされたエンティティ名、あるいは属性のデータ型を検索したい、という状況です。また、実行時にプログラムでモデルに修正を施すこともあるでしょう（これは当該モデルを使う前に実施しなければなりません。`NSManagedObjectModel`を参照）。実行時に管理オブジェクトモデルにアクセスする手段はいくつかあります。まず、永続スタックを介して、永続ストアコーディネータからモデルを取得できます。したがって、管理オブジェクトコンテキストからモデルを取得するためには、次のようなコードを記述すればよいことになります。

```
[[<#A managed object context#> persistentStoreCoordinator] managedObjectModel];
```

また、モデルはエンティティ記述からも検索できるので、管理オブジェクトがあれば、次の例のように、そのエンティティ記述を介してモデルを検索可能です。

```
[[<#A managed object#> entity] managedObjectModel];
```

場合によっては、モデルの「直接」参照、すなわちモデルそのものを返すメソッドが使えることもあります。`NSPersistentDocument`には`managedObjectModel`というメソッドがあるので、文書の管理オブジェクトコンテキストが使う、永続ストアコーディネータに関連付けられたモデルを取得できます。Core DataのApplicationテンプレートを使っていれば、アプリケーションデリゲートが、モデルの参照を管理しています。

フェッチ要求テンプレートをプログラムで作成

フェッチ要求テンプレートをプログラムで実行時に作成し、`setFetchRequestTemplate:forName:`で、モデルに関連付けることができます（リスト 1を参照）。ただし、モデルを修正できるのは、ストアコーディネータがそれを使い始める前に限ることに注意してください。

リスト 1　フェッチ要求テンプレートをプログラムで作成

```
NSManagedObjectModel *model = <#Get a model#>;
NSFetchRequest *requestTemplate = [[NSFetchRequest alloc] init];
NSEntityDescription *publicationEntity =
    [[model entitiesByName] objectForKey:@"Publication"];
[requestTemplate setEntity:publicationEntity];

NSPredicate *predicateTemplate = [NSPredicate predicateWithFormat:
    @"(mainAuthor.firstName like[cd] $FIRST_NAME) AND \
        (mainAuthor.lastName like[cd] $LAST_NAME) AND \
        (publicationDate > $DATE)"];
[requestTemplate setPredicate:predicateTemplate];

[model setFetchRequestTemplate:requestTemplate
    forName:@"PublicationsForAuthorSinceDate"];
[requestTemplate release];
```

フェッチ要求テンプレートへのアクセス

フェッチ要求テンプレートを検索して使うコード例を「実行時に管理オブジェクトモデルにアクセスし使用する方法」に示します。置換辞書には、テンプレートに定義されている変数に対応するキーが、すべて記述されていなければなりません。変数値として**null**を与えたい場合は、`NSNull`オブジェクトを使ってください（「述語の使い方」を参照）。

リスト 2　フェッチ要求テンプレートの使い方

```
NSManagedObjectModel *model = <#Get a model#>;
NSError *error = nil;
NSDictionary *substitutionDictionary = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Fiona", @"FIRST_NAME", @"Verde", @"LAST_NAME",
    [NSDate dateWithTimeIntervalSinceNow:-31356000], @"DATE", nil];
NSFetchRequest *fetchRequest =
    [model fetchRequestFromTemplateWithName:@"PublicationsForAuthorSinceDate"
            substitutionVariables:substitutionDictionary];
NSArray *results =
    [aManagedObjectContext executeFetchRequest:fetchRequest error:&error];
```

テンプレートに置換変数がない場合は、次のいずれかの方法で対処してください。

1. `fetchRequestFromTemplateWithName:substitutionVariables:`に、変数引数として`nil`を渡して呼び出し。

2. `fetchRequestTemplateForName:`を実行し、その結果を `copy`。

   `fetchRequestTemplateForName:`で取得したフェッチ要求をそのまま（コピーせずに）使うと、例外（「`Can't modify a named fetch request in an immutable model`」）が発生します。

管理オブジェクトモデルのローカライズ

管理オブジェクトモデルは、エンティティ名、プロパティ名、エラーメッセージなど、さまざまな箇所がローカライズ可能になっています。重要なのは、「自国語版の開発」もローカライズの一種である、ということです。外国語版を提供する予定がなくても、エラーメッセージにコンピュータでの処理に結び付いた名前ではなく「自然言語」に基づく名前が使われていれば、ユーザにとって分かりやすいでしょう（たとえば「firstNameは必須プロパティです」ではなく「ファーストネームは必須プロパティです」）。

モデルのローカライズは、次の表に示すようなパターンの、ローカライズ辞書を作成して行います。

表 1　管理オブジェクトモデルのローカライズ辞書におけるキーと値

キー	値	注
"Entity/NonLocalizedEntityName"	"LocalizedEntityName"	
"Property/NonLocalizedPropertyName/Entity/EntityName"	"LocalizedPropertyName"	1
"Property/NonLocalizedPropertyName"	"LocalizedPropertyName"	
"ErrorString/NonLocalizedErrorString"	"LocalizedErrorString"	

注：（1）異なるエンティティに属する、非ローカライズ名が同じプロパティに、異なるローカライズ名を与えたい場合

ローカライズ辞書には`localizationDictionary`メソッドでアクセスできます。ただしMac OS X v10.4では、Core Dataが自分自身で使う（たとえばエラーメッセージをローカライズして報告する）ために辞書を遅延読み込みするまでの間、`localizationDictionary`が`nil`を返す場合があります。

文字列ファイル

モデルをローカライズする方法として最も簡単なのは、対応する文字列ファイルを作るというものです。文字列ファイルの名前は、モデルファイル名の拡張子「.xcdatamodel」を「.strings」に変更したものになります（たとえばモデルファイル名が「MyDocument.xcdatamodel」ならば文字列ファイル名は「MyDocumentModel.strings」。なお、モデルファイル名の末尾が「Model」となっている場合、文字列ファイル名には重ねて「Model」という字句が追加されるので、モデルファイル名が「JimsModel.xcdatamodel」であれば、少々不恰好ではありますが、文字列ファイル名は「JimsModelModel.strings」となります）。ファイル形式は、ローカライズに使う標準的な文字列ファイルと同様です（「文字列リソースのローカライズ」を参照）。ただし、キーと値のパターンは、表 1 （37 ページ）に示したようになります。

「従業員」エンティティが使われているモデルの文字列ファイルには、たとえば次のような記述があるでしょう。

```
"Entity/Emp" = "Employee";
"Property/firstName" = "First Name";
"Property/lastName" = "Last Name";
"Property/salary" = "Salary";
```

ローカライズ辞書をプログラムで設定する手順

ローカライズ辞書を実行時に設定するためには、`NSManagedObjectModel`の`setLocalizationDictionary:`メソッドを使います。表 1 （37 ページ）に示したようなキーと値を記述した辞書を作成し、モデルに関連付ける必要があります。この処理は、モデルを使って管理オブジェクトをフェッチまたは生成する前に実行しなければなりません。モデルが編集できなくなってしまうからです。リスト 3 （38 ページ）に、プログラムで管理オブジェクトモデルを生成し、ローカライズ辞書を反映させる例を示します。エンティティ名は「Run」で、実行時にはRunクラスで表されます。このエンティティには、「date」と「processID」という、日付型および整数型の、2つの属性があります。プロセスIDには、非負でなければならない、という制約があります。

リスト 3　管理オブジェクトモデルをプログラムで生成

```
NSManagedObjectModel *mom = [[NSManagedObjectModel alloc] init];
NSEntityDescription *runEntity = [[NSEntityDescription alloc] init];
[runEntity setName:@"Run"];
[runEntity setManagedObjectClassName:@"Run"];
[mom setEntities:[NSArray arrayWithObject:runEntity]];
[runEntity release];

NSMutableArray *runProperties = [NSMutableArray array];

NSAttributeDescription *dateAttribute = [[NSAttributeDescription alloc] init];
[runProperties addObject:dateAttribute];
[dateAttribute release];
[dateAttribute setName:@"date"];
[dateAttribute setAttributeType:NSDateAttributeType];
[dateAttribute setOptional:NO];

NSAttributeDescription *idAttribute= [[NSAttributeDescription alloc] init];
[runProperties addObject:idAttribute];
[idAttribute release];
[idAttribute setName:@"processID"];
[idAttribute setAttributeType:NSInteger32AttributeType];
[idAttribute setOptional:NO];
[idAttribute setDefaultValue:[NSNumber numberWithInt:0]];

NSPredicate *validationPredicate = [NSPredicate predicateWithFormat:@"SELF >= 0"];
NSString *validationWarning = @"Process ID < 0";
[idAttribute setValidationPredicates:[NSArray arrayWithObject:validationPredicate]
    withValidationWarnings:[NSArray arrayWithObject:validationWarning]];

[runEntity setProperties:runProperties];

NSMutableDictionary *localizationDictionary = [NSMutableDictionary dictionary];
[localizationDictionary setObject:@"Process ID"
    forKey:@"Property/processID/Entity/Run"];
[localizationDictionary setObject:@"Date"
    forKey:@"Property/date/Entity/Run"];
[localizationDictionary setObject:@"Process ID must not be less than 0"
    forKey:@"ErrorString/Process ID < 0"];
[mom setLocalizationDictionary:localizationDictionary];
```

管理オブジェクト

この章では、管理オブジェクトとは何か、どのようにデータを保存するか、カスタム管理オブジェクトクラスはどうやって実装するか、オブジェクトライフサイクルに関する問題点、フォールティングなどについて、基本的な説明をします。管理オブジェクトの使い方については、『*Core Data Programming Guide*』の次の章でも解説しています。

- 「管理オブジェクトの生成と削除」（55 ページ）
- 「管理オブジェクトのフェッチ」（61 ページ）
- 「管理オブジェクトの使い方」（65 ページ）

基本機能

管理オブジェクトは、`NSManagedObject`またはそのサブクラスのインスタンスで、エンティティのインスタンスを表します。`NSManagedObject`は総称クラスで、管理オブジェクトに必要な基本機能がすべて実装されています。`NSManagedObject`のカスタムサブクラスを作成することもできますが、必須ではありません。対応するエンティティにカスタムロジックを組み込む必要がなければ、当該エンティティに対応するカスタムクラスは、なくても構わないのです。これが有用なのは、たとえば、カスタムアクセサメソッドやカスタム検証メソッドを提供したい、標準以外の属性を扱いたい、依存キーを指定したい、ほかの属性値をもとに計算で値を求めたい、その他のカスタムロジックを実装したい、などといった場合です。

管理オブジェクトはエンティティ記述（`NSEntityDescription`のインスタンス）に関連付けられています。これは、オブジェクトに関するメタデータ（オブジェクトが表現するエンティティの名前、属性や関係の名前など）を設定したものです。

管理オブジェクトは、管理オブジェクトコンテキスト（あるいは単に「コンテキスト」）にも関連付けられています。コンテキストの側から見ると、管理オブジェクトは永続ストアに格納されたレコードの表現を与えていることになります。ひとつのコンテキスト内では、永続ストアのレコードに対応する管理オブジェクトは1つに限りますが、コンテキストが複数あれば、レコードの管理オブジェクトによる表現がそれぞれ存在しても構いません。すなわち、管理オブジェクトとそれが表現するデータレコードの間には対1の関係があるけれども、レコードと対応する管理オブジェクトの間には対多の関係がある、ということです。

プロパティとデータストレージ

見方によっては、`NSManagedObject`は辞書のように振る舞います。総称的なコンテナオブジェクトで、関連する`NSEntityDescription`オブジェクトで定義されるプロパティの、効率的なストレージ機能を提供します。`NSManagedObject`には、文字列、日付、数値など、属性値によく使われる型に関する支援機能が組み込まれています（詳しくは`NSAttributeDescription`を参照）。したがって、

通常、サブクラスにインスタンス変数を定義する必要はありません。大容量のバイナリデータオブジェクトを扱う場合には、処理性能に若干配慮する必要があります（「大容量データオブジェクト（BLOB）」（140 ページ）を参照）。

標準以外の属性

NSManagedObjectには、文字列、日付、数値など、属性値によく使われる型に関する支援機能が組み込まれています（詳しくはNSAttributeDescriptionを参照）。デフォルトでは、NSManagedObjectはプロパティをオブジェクトの形で内部的に保存するようになっています。Core Dataは一般に、カスタムインスタンス変数を定義して用いるよりも、自ら管理するストレージを使う方が効率的に動作します。

しかし、直接の支援機能がない、色、Cの構造体などといった型を使いたい場合もあるでしょう。たとえばグラフィックスアプリケーションでは、Rectangleというエンティティを定義し、色や境界を表す属性として、NSColorやNSRectという構造体のインスタンスを使う、ということが考えられます。これを実装するためには、NSManagedObjectのサブクラスを定義する必要があります（「非標準の永続属性型」（89 ページ）を参照）。

日付と時刻

NSManagedObjectでは、日付属性をNSDateオブジェクトで表し、内部的には基準日（時間帯はGMT）以降のNSTimeInterval値として保存しています。時間帯は明示的に保存していないので、Core Dataの日付属性は必ずGMTで表すようにしてください。データベースにおける検索も、これに従って正規化されます。時間帯情報を保持しておきたい場合は、独自のモデルを用意し、時間帯属性を別に保存するようにしなければなりません。このような場合にも、NSManagedObjectのサブクラスを作成する必要があります。

カスタム管理オブジェクトクラス

NSManagedObjectには、管理オブジェクトモデルのエンティティ記述と連携する形で、プロパティや値の検証支援など、豊富な機能が実装されています。それでも、NSManagedObjectのサブクラスを定義し、カスタム機能を実装したい状況は多々あるでしょう。しかし、サブクラスを作成するに当たって、避けるべき事項がいくつかあります。また、Core Dataがモデル化プロパティのライフサイクルを管理している、という点も重要です。

メソッドのオーバーライド

NSManagedObjectクラスは、管理オブジェクトをCore Dataインフラストラクチャにうまく組み込めるよう、NSObjectの機能の多くをカスタマイズしています。Core DataはNSManagedObjectの次のメソッドの実装に依存しているので、オーバーライドしないでください:primitiveValueForKey:、setPrimitiveValue:forKey:、isEqual:、hash、superclass、class、self、zone、isProxy、isKindOfClass:、isMemberOfClass:、conformsToProtocol:、respondsToSelector:、retain、release、autorelease、retainCount、managedObjectContext、entity、objectID、isInserted、isUpdated、isDeleted、isFault。さらにdescriptionメソッドも、オーバーライドはお勧めできません。デバッグ中にこのメソッドがフォールトを発動した場合の結果は予測でき

ないからです。initWithEntity:insertIntoManagedObjectContext:も同様です。また、valueForKey:、setValue:forKeyPath:などといったキー値コーディング用のメソッドも、通常はオーバーライドしないでください。

オーバーライドするべきではないメソッドを挙げましたが、オーバーライドして構わないメソッドの中にも、スーパークラスの実装をまず呼び出してから、独自の処理を記述しなければならないものもあります。awakeFromInsertや、awakeFromFetch、あるいはvalidateForUpdate:をはじめとする検証メソッドがこれに当たります。

モデル化プロパティ

Mac OS X v10.5以降では、属性や関係の取得/設定メソッドが動的に生成され、パブリックおよびプリミティブなメソッドとして効率的に使えます。その対象は、管理オブジェクトに対応する管理オブジェクトモデルの、エンティティで定義されたプロパティです。したがって、通常、モデル化プロパティのカスタムアクセサメソッドは、別途記述する必要がありません。

管理オブジェクトのサブクラスでは、モデル化属性に対応するプロパティをインターフェイスファイルに宣言できますが、インスタンス変数は宣言できません。

```
@interface MyManagedObject : NSManagedObject {
}
@property (nonatomic, retain) NSString *title;
@property (nonatomic, retain) NSDate *date;
@end
```

ここで、プロパティがnonatomicかつretainと宣言されていることに注意してください。性能上の理由で、Core Dataは通常、オブジェクト値をコピーしません。値のクラスがNSCopyingプロトコルに準拠したものであっても同様です。

実装ファイルでは、プロパティをdynamicと指定します。

```
@implementation MyManagedObject
@dynamic title;
@dynamic date;
@end
```

モデル化プロパティのライフサイクルはCore Dataが管理しているので、参照カウントを管理している環境では、*モデル化*プロパティをdeallocで解放してはなりません（管理オブジェクトモデルで指定せずに独自のプロパティを追加した場合は、通常のCocoaの規則を適用）。

カスタムアクセサメソッドを実装する場合、準拠するべき実装パターンがいくつかあります。「管理オブジェクトのアクセサメソッド」（45 ページ）を参照してください。

オブジェクトのライフサイクル: 初期化と割り当て解除

重要なのは、管理オブジェクトのライフサイクルは、Core Data が「管理する」ものであることです。フォールティングや取り消し管理の機能があるため、管理オブジェクトのライフサイクルについて、標準的なCocoaオブジェクトと同じように扱うことはできません。管理オブジェクトのインスタンス生成、破棄、復元は、必要に応じてフレームワークが行います。

管理オブジェクトを生成する際、その初期化には、管理オブジェクトモデルでエンティティに指定されたデフォルト値が使われます。多くの場合、モデルにデフォルト値を指定しておくだけで、初期化には充分です。しかし場合によっては、初期化のために追加の処理が必要になるでしょう。モデルでは表現できない、動的に決まる値（現在の日付や時刻など）で初期化するような状況です。

典型的なCocoaのクラスでは、初期化を行うメソッド（多くの場合init）をオーバーライドするのが普通です。それに対し、NSManagedObjectのサブクラスには、初期化の処理をカスタマイズする方法が3つあります。initWithEntity:insertIntoManagedObjectContext:、awakeFromInsert、awakeFromFetchのいずれかをオーバーライドする、というものです。initはオーバーライドしないでください。また、initWithEntity:insertIntoManagedObjectContext:をオーバーライドする方法も、あまりお勧めできません。このメソッド内で状態を変更しても、取り消し/再実行の処理に、適切に反映されないからです。残り2つのメソッド、awakeFromInsertとawakeFromFetchを、状況によって使い分けてください。

- awakeFromInsertは、オブジェクトのライフサイクルを通して1回だけ、生成時に呼び出されます。

  awakeFromInsertが起動されるのは、initWithEntity:insertIntoManagedObjectContext:またはinsertNewObjectForEntityForName:inManagedObjectContext:を起動した直後です。awakeFromInsertでは、下記のコード例のように、オブジェクトの生成日付など、デフォルトで定義されている特別なプロパティの値も初期化できます。

  ```
  - (void) awakeFromInsert
  {
      [super awakeFromInsert];
      [self setCreationDate:[NSDate date]];
  }
  ```

- awakeFromFetchは、永続ストアから（フェッチして）オブジェクトを再初期化する際に起動されます。

  awakeFromFetchをオーバーライドして、たとえば、一時値その他のキャッシュを設けることができます。awakeFromFetchの実行中、変更処理は明示的に無効にされるので、パブリックな設定アクセサメソッドを使っても、オブジェクトやそのコンテキストを汚染することはありません。しかし、まさにこの理由で、関係は操作するべきではありません。関係先のオブジェクトに、変更が正しく伝わらないからです。これが問題になる場合は、代わりにawakeFromInsertをオーバーライドするか、performSelector:withObject:afterDelay:など、イベントループ関連のいずれかのメソッドを利用してください。

一般に、一時プロパティその他の変数をクリアする目的で、deallocやfinalizeをオーバーライドしてはいけません。このような場合はdidTurnIntoFaultをオーバーライドしてください。didTurnIntoFaultは、オブジェクトがフォールト状態になったときや、実際に割り当て解除する直前に、自動的に呼び出されます。管理オブジェクトをわざとフォールト状態にして、メモリのオーバーヘッドを削減することもある（「メモリ消費量のオーバーヘッド削減」（139 ページ）を参照）ので、クリーンアップ処理はdidTurnIntoFaultで適切に実行することが大切です。

検証

NSManagedObjectには、個々のプロパティ値や、プロパティどうしの値の組み合わせを検証できるよう、一貫してフックが用意されています。通常は、validateValue:forKey:error:をオーバーライドするのではなく、validate<Key>:error:の形のメソッドを、NSKeyValueCodingプロトコルに従って実装してください。プロパティどうしの値の組み合わせを検証したい場合は、validateForUpdate:や、関係する検証メソッドをオーバーライドします。

カスタムプロパティ検証メソッドから、validateValue:forKey:error:を呼び出してはなりません。これが呼び出された時点で、無限ループに陥ってしまいます。カスタム検証メソッドを実装する場合も同様に、直接呼び出さないようにしてください。適切なキーを指定して、validateValue:forKey:error:を呼び出すようにしてください。これにより、管理オブジェクトモデルで定義された制約が、きちんと適用されます。

プロパティ値の組み合わせを検証するカスタムメソッド（validateForUpdate:など）を実装する場合は、その先頭で、スーパークラスの実装を呼び出してください。これにより、関係する個々のプロパティの検証メソッドが呼び出されるようになります。検証の結果、複数箇所にエラーが見つかった場合は、その情報を配列にまとめ、NSDetailedErrorsKeyというキーで、NSErrorオブジェクトのユーザ情報辞書に追加したものを返してください。

フォールティング

管理オブジェクトは、通常、永続ストアに格納されている、あるデータを表現しています。管理オブジェクトが「フォールト」であるとは、そのプロパティ値を外部データストアからまだ読み込んでいない状態のことです（詳しくは「フォールティングと一意化」（105 ページ）を参照）。永続プロパティ値にアクセスした時点でフォールトが「発動」し、自動的にストアからデータを検索するようになっています。これは（場合によっては永続ストアとの間で何度も往復しなければならない）比較的高価な処理なので、必要もないのに発動することは避けなければなりません（「フォールティングの動作」（136 ページ）を参照）。

descriptionメソッドを呼び出したためにフォールトが発動することはありませんが、descriptionというカスタムメソッドを実装し、その中でオブジェクトの永続プロパティにアクセスしていれば、発動する場合があります。したがって、このような形でdescriptionをオーバーライドすることは避けてください。

管理オブジェクトの個々の属性値を、必要が生じた時点で読み込むようにする手段はありません。大容量の属性を操作するためのパターンについては、「大容量データオブジェクト（BLOB）」（140 ページ）を参照してください。

管理オブジェクトのアクセサメソッド

この章では、管理オブジェクトのカスタムアクセサメソッドを実装することが望ましい状況を説明し、属性や関係のアクセサメソッドを実装する方法を示します。また、プリミティブなアクセサメソッドの実装方法も解説します。

概要

Mac OS X v10.5では、管理オブジェクトクラスの属性や関係の取得/設定メソッドが動的に生成され、パブリックおよびプリミティブなメソッドとして効率的に使えます。したがって通常、管理オブジェクトのプロパティで、対応する管理オブジェクトモデルのエンティティで定義されているものについては、アクセサメソッドを別途記述する必要がありません。もっとも、Objective-Cの宣言済みプロパティ機能を使ってプロパティを宣言し、コンパイラの警告を抑制することは、状況によっては有用かもしれません。処理性能に優れ、型検査の機能も活用できるので、アクセサメソッドを直接使うようお勧めします。もっとも、これはキー値コーディング（KVC; Key-Value Coding）にも準拠しているので、必要ならば`valueForKey:`などの標準的なキー値コーディング用メソッドも使えます。一時プロパティを使って標準以外のデータ型を扱いたい（「非標準の永続属性型」（89 ページ）を参照）場合や、スカラのインスタンス変数を使って属性を表したい場合は、カスタムアクセサメソッドを記述する必要があります。

カスタム実装

`NSManagedObject`のサブクラスでは、アクセサメソッドの実装方法がほかのクラスとは異なります。

- カスタムインスタンス変数を用いないのであれば、内部ストアとの間でプロパティ値を検索、保存するために、プリミティブなアクセサメソッドが使えます。
- 状況に応じて、適切なアクセス通知メソッド、変更通知メソッドを起動してください（`willAccessValueForKey:`、`didAccessValueForKey:`、`willChangeValueForKey:`、`didChangeValueForKey:`、`willChangeValueForKey:withSetMutation:usingObjects:`、`didChangeValueForKey:withSetMutation:usingObjects:`）。

  `NSManagedObject`はモデル化プロパティについて、KVO（キー値監視; Key-Value Observing）の自動変更通知機能を無効にします。プリミティブなアクセサメソッドが、アクセス通知メソッドや変更通知メソッドを呼び出すことはありません。非モデル化プロパティについては、Mac OS X v10.4でも、KVOの自動変更通知機能は無効になります。Mac OS X v10.5以降では、Core Dataは`NSObject`の振る舞いを取り入れています。
- エンティティモデルで定義されていないプロパティのアクセサメソッドでは、自動変更通知機能を有効にしても、あるいは適当な変更通知メソッドを呼び出すようにしても構いません。

Xcodeのデータモデリングツールを使えば、どのモデル化プロパティについても、アクセサメソッドの実装コードを生成できます。

キー値コーディングのアクセスパターン

キー値コーディングが管理オブジェクトに対して採用しているアクセスパターンは、`NSObject`のサブクラスの場合とおおむね同じです（`valueForKey:`を参照）。ただし、通常の手順でアクセスできるか検査した結果、`valueForKey:`が「**unbound key**」（束縛されていないキー）例外を投げた場合には、`NSManagedObject`のキー値コーディング（KVC）メカニズムにより、キーが適切にモデル化されているかの確認が行われます。キーがエンティティのプロパティに合致した場合、まず「`primitive`*Key*」という形の名前のアクセサメソッドを検索し、見つからなければ、管理オブジェクトの内部ストレージから、*key*の値を検索するようになっています。これに失敗した場合、`NSManagedObject`は「**unbound key**」例外を投げます（`valueForKey:`と同じ動作）。

動的に生成されるアクセサメソッド

デフォルトでは、管理オブジェクトクラスのモデル化プロパティ（属性および関係）について取得/設定メソッドが動的に生成され、パブリックおよびプリミティブなメソッドとして効率的に使えます。`add<Key>Object:`や`remove<Key>s:`のような、キー値コーディングにおけるミュータブルなプロキシのメソッドもその対象です（詳しくは`mutableSetValueForKey:`を参照）。管理オブジェクトは、実際上、対多の関係すべてに関するミュータブルなプロキシです。

> **注：** 独自にアクセサを実装すれば、動的に生成されたメソッドがこれに置き換わることはありません。

たとえば属性`firstName`を持つエンティティに対しては、`firstName`、`setFirstName:`、`primitiveFirstName`、`setPrimitiveFirstName:`というメソッドが自動生成されます。*これはエンティティが`NSManagedObject`で表される場合も同じです*。このメソッドを起動したときにコンパイラの警告が出ないようにするため、Objective-C 2.0の宣言済みプロパティ機能を利用するとよいでしょう（「宣言」（47 ページ）を参照）。

自動生成されるプロパティのアクセサメソッドは、`(nonatomic, retain)`という指定（*一般にも推奨される設定*）がついたものになります。`nonatomic`であるのは、アクセサはアトミックでない方が効率がよいからです。また、一般にアクセサメソッドのレベルでは、Core Dataアプリケーションでスレッドセーフであることを保証できません（マルチスレッド環境でのCore Dataの使い方については「Concurrency with Core Data」（131 ページ）を参照）。

動的プロパティは、常に`nonatomic`であることに加え、`retain`または`copy`属性を指定しなければなりません。`assign`を指定しても、`retain`として扱われます。`copy`を指定するとオーバーヘッドが増すので、できるだけ避けてください。関係に`copy`を指定することはできません。`NSManagedObject`は`NSCopying`プロトコルに従っておらず、また、対多関係の動作には適していないからです。

重要： また、対1関係に対して`copy`を指定しても、*実行時*にエラーになります。

宣言

Objective-C 2.0のプロパティを使って、管理オブジェクトクラスのプロパティを宣言できます。これは、Core Dataが生成するデフォルトのアクセサを使っても、コンパイラの警告が出ないようにするための方法です。*宣言を生成する最も簡単な方法は、Xcodeモデリングツールで関係を選択し、「設計(Design)」＞「データモデル(Data Model)」＞「Obj-C 2.0 Method Declarationsをクリップボードにコピー(Copy Obj-C 2.0 Method Declarations to Clipboard)」を実行後、*必要に応じてコードを修正するというものです。

属性や関係を宣言する方法は、ほかのオブジェクトのプロパティを宣言する場合と同じです（以下の例を参照）。対多関係を宣言する場合、プロパティの型は「`NSSet *`」となります（取得アクセサの戻り値は、KVOに準拠したミュータブルなプロキシ*ではありません*。詳しくは「対多の関係」（66 ページ）を参照）。

```
@interface Employee : NSManagedObject
{ }
@property(nonatomic, retain) NSString* firstName, lastName;
@property(nonatomic, retain) Department* department;
@property(nonatomic, retain) Employee* manager;
@property(nonatomic, retain) NSSet* directReports;
@end
```

カスタムクラスを使わないのであれば、プロパティを`NSManagedObject`のカテゴリで宣言しても、コンパイラの警告を抑制できます。

```
@interface NSManagedObject (EmployeeAccessors)

@property(nonatomic, retain) NSString* firstName, lastName;
@property(nonatomic, retain) Department* department;
@property(nonatomic, retain) Employee* manager;
@property(nonatomic, retain) NSSet* directReports;
@end
```

自動生成される対多関係のミューテータメソッドについても、同じ方法でコンパイラの警告を抑制できます（次の例を参照）。

```
@interface Employee (DirectReportsAccessors)

- (void)addDirectReportsObject:(Employee *)value;
- (void)removeDirectReportsObject:(Employee *)value;
- (void)addDirectReports:(NSSet *)value;
- (void)removeDirectReports:(NSSet *)value;

@end
```

通常は属性を保持（retain）しますが、属性クラスにミュータブルなサブクラスがあり、これが`NSCopying`プロトコルを実装しているならば、カプセル化を維持するため、次の例のように`copy`を指定しても構いません。

```
@property(nonatomic, copy) NSString* firstName, lastName;
```

実装

実装の指定には、次の例のように@dynamicキーワードを使います。もっとも、何も指定しなければ、@dynamicが指定されたと見なされるので、このキーワードはなくても構いません。

```
@dynamic firstName, lastName;
@dynamic department, manager;
@dynamic directReports;
```

スカラ値を扱えるようにしたい場合を除き、独自にメソッドを実装する必要はないはずです。実行時にCore Dataが生成するメソッドは、独自に実装したものよりも効率的です。

継承

NSManagedObjectのサブクラスが2つあって、親クラスで動的プロパティを実装し、そのサブクラス（NSManagedObjectから見ると孫に当たるクラス）で当該プロパティ用のメソッドをオーバーライドしている場合、ここからsuperを呼び出すことはできません。

```
@interface Parent : NSManagedObject
@property(nonatomic, retain) NSString* parentString;
@end

@implementation Parent
@dynamic parentString;
@end

@interface Child : Parent
@end

@implementation Child
- (NSString *)parentString
{
    // 「selector not found」例外が発生
    return parentString.foo;
}
@end
```

属性や対1関係のカスタムアクセサメソッド

重要： 標準的な、あるいはプリミティブなアクセサメソッドは、独自に実装せず、できる限り動的プロパティ（実装を@dynamicとして指定したプロパティ）を使うようにしてください。

独自の属性や対1関係のアクセサメソッドを実装する場合、プリミティブなアクセサメソッドを使って、管理オブジェクトが持つプライベートな内部ストアから値を取得し、あるいはここに値を設定することになります。このとき、適切なアクセス通知メソッド、変更通知メソッドを起動しなければなりません（リスト 1 （49 ページ）を参照）。NSManagedObjectに実装された、プリミティブな設定アクセサには、メモリ管理機能も組み込まれています。

リスト 1　　カスタム管理オブジェクトクラスの実装: 属性アクセサメソッドの例

```
@interface Department : NSManagedObject
{
}
@property(nonatomic, retain) NSString *name;
@end

@interface Department (PrimitiveAccessors)
- (NSString *)primitiveName;
- (void)setPrimitiveName:(NSString *)newName;
@end


@implementation Department

@dynamic name;

- (NSString *)name
{
    [self willAccessValueForKey:@"name"];
    NSString *myName = [self primitiveName];
    [self didAccessValueForKey:@"name"];
    return myName;
}

- (void)setName:(NSString *)newName
{
    [self willChangeValueForKey:@"name"];
    [self setPrimitiveName:newName];
    [self didChangeValueForKey:@"name"];
}
@end
```

デフォルトの実装では、属性値をコピーしないようになっています。（たとえばNSStringのように）属性値がミュータブルで、NSCopyingプロトコルを実装している場合は、カスタムアクセサで値をコピーすることにより、（たとえばNSMutableStringのインスタンスが値として渡される場合に）カプセル化を維持することができます。その例をリスト 2 （49 ページ）に示します。この例では、（例を示す目的上）取得アクセサを実装していないことにも注意してください。したがってCore Dataが自動的に生成します。

リスト 2　　カスタム管理オブジェクトクラスの実装: 値をコピーする設定アクセサの例

```
@interface Department : NSManagedObject
{
}
@property(nonatomic, copy) NSString *name;
@end

@implementation Department

@dynamic name;

- (void)setName:(NSString *)newName
{
    [self willChangeValueForKey:@"name"];
    // NSStringはNSCopyingを実装しているので、属性値をコピーする
    NSString *newNameCopy = [newName copy];
```

```
    [self setPrimitiveName:newNameCopy];
    [newNameCopy release];
    [self didChangeValueForKey:@"name"];
}
@end
```

属性をスカラ型（NSInteger、CGFloatなど）で表す、あるいはNSKeyValueCodingでサポートされるいずれかの構造体（NSRect、NSPoint、NSSize、NSRangeなど）で表す場合、アクセサメソッドはリスト 3 （50 ページ）に示すように実装しなければなりません。そのほかの属性型を使う場合は、別のパターンになります（「非標準の永続属性型」 （89 ページ）を参照）。

リスト 3　　カスタム管理オブジェクトクラスの実装: スカラ属性値の例

```
@interface Circle : NSManagedObject
{
    CGFloat radius;
}
@property CGFloat radius;
@end

@implementation Circle

- (CGFloat)radius
{
    [self willAccessValueForKey:@"radius"];
    float f = radius;
    [self didAccessValueForKey:@"radius"];
    return f;
}

- (void)setRadius:(CGFloat)newRadius
{
    [self willChangeValueForKey:@"radius"];
    radius = newRadius;
    [self didChangeValueForKey:@"radius"];
}
@end
```

対多関係のカスタムアクセサメソッド

重要： 標準的な、あるいはプリミティブなアクセサメソッドは、独自に実装せず、できる限り動的プロパティ（実装を@dynamicとして指定したプロパティ）を使うようにしてください。

通常、対多関係には、mutableSetValueForKey:を使ってアクセスします。その戻り値は、関係を変化させ、適切なKVO（キー値監視）通知を送信する、プロキシオブジェクトです。対多関係をコレクションの形で返すアクセサメソッドを、独自に実装しなければならないことはほとんどありません。しかし、独自の実装がある場合、永続関係を表すコレクションに修正を施すと、フレームワークは（add<Key>Object:、remove<Key>Object:などの）ミューテータメソッドを呼び出すようになっています（フェッチ済みプロパティには、コレクションの形で取得/設定する、ミュータブルなアクセサメソッドがありません）。これを正常に動作させるためには、add<Key>Object:とremove<Key>Object:の組、add<Key>:とremove<Key>:の組、あるいはこの両方を実装する必要が

あります。これ以外の取得アクセサ（countOf<Key>:、enumeratorOf<Key>:、memberOf<Key>:など）を実装し、コード内で使うこともできますが、フレームワークから呼び出されるという保証はありません。

重要： 性能上の理由で、mutableSetValueForKey:を呼び出したときに管理オブジェクトから返されるプロキシオブジェクトには、関係を対象とするset<Key>:型の設定アクセサがありません。たとえば、Departmentクラスにemployeesという対多関係があり、アクセサメソッドemployeesおよびsetEmployees:を実装している場合に、mutableSetValueForKey:@"employees"の戻り値であるプロキシオブジェクトを使って関係を操作しても、setEmployees:は起動*されません*。代わりに、ほかのミュータブルなプロキシアクセサを実装し、オーバーライドしてください。

コレクションの形で取得/設定する、モデル化プロパティのアクセサを実装する場合、適切なKVO通知メソッドを呼び出さなければなりません。リスト 4 （51 ページ）に、対多関係（Departmentクラスのemployees）のアクセサメソッドを実装した例を示します。*実装を生成する最も簡単な方法は、Xcodeモデリングツールで関係を選択し、「設計(Design)」＞「データモデル(Data Model)」＞「Obj-C 2.0 Method {Declarations/Implementations}をクリップボードにコピー(Copy Obj-C 2.0 Method {Declarations/Implementations} to Clipboard)」を実行するというものです。*

リスト 4 管理オブジェクトクラス: 対多関係のカスタムアクセサを実装した例

```
@interface Department : NSManagedObject
{
}
@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSSet *employees;
@end


@interface Department (DirectReportsAccessors)

- (void)addEmployeesObject:(Employee *)value;
- (void)removeEmployeesObject:(Employee *)value;
- (void)addEmployees:(NSSet *)value;
- (void)removeEmployees:(NSSet *)value;

- (NSMutableSet*)primitiveEmployees;
- (void)setPrimitiveEmployees:(NSMutableSet*)value;

@end


@implementation Department

@dynamic name;
@dynamic employees;

- (void)addEmployeesObject:(Employee *)value
{
    NSSet *changedObjects = [[NSSet alloc] initWithObjects:&value count:1];

    [self willChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueUnionSetMutation
          usingObjects:changedObjects];
    [[self primitiveEmployees] addObject:value];
    [self didChangeValueForKey:@"employees"
```

```
            withSetMutation:NSKeyValueUnionSetMutation
            usingObjects:changedObjects];

    [changedObjects release];
}

- (void)removeEmployeesObject:(Employee *)value
{
    NSSet *changedObjects = [[NSSet alloc] initWithObjects:&value count:1];

    [self willChangeValueForKey:@"employees"
            withSetMutation:NSKeyValueMinusSetMutation
            usingObjects:changedObjects];
    [[self primitiveEmployees] removeObject:value];
    [self didChangeValueForKey:@"employees"
            withSetMutation:NSKeyValueMinusSetMutation
            usingObjects:changedObjects];

    [changedObjects release];
}

- (void)addEmployees:(NSSet *)value
{
    [self willChangeValueForKey:@"employees"
            withSetMutation:NSKeyValueUnionSetMutation
            usingObjects:value];
    [[self primitiveEmployees] unionSet:value];
    [self didChangeValueForKey:@"employees"
            withSetMutation:NSKeyValueUnionSetMutation
            usingObjects:value];
}

- (void)removeEmployees:(NSSet *)value
{
    [self willChangeValueForKey:@"employees"
            withSetMutation:NSKeyValueMinusSetMutation
            usingObjects:value];
    [[self primitiveEmployees] minusSet:value];
    [self didChangeValueForKey:@"employees"
            withSetMutation:NSKeyValueMinusSetMutation
            usingObjects:value];
}
```

プリミティブなアクセサメソッド

プリミティブなアクセサメソッドも、キー値コーディングに準拠した、「通常の」（すなわちパブリックな）アクセサメソッドと同様です。ただし、Core Dataはこれを、データにアクセスする最も基本的なデータメソッドとして扱います。したがって、キー値アクセス通知やキー値監視通知は発行*しません*。`primitiveValueForKey:`および`setPrimitiveValue:forKey:`の動作は、パブリックなアクセサメソッドである`valueForKey:`および`setValue:forKey:`に対応します。

通常、プリミティブなアクセサメソッドを実装する必要は、ほとんどありません。しかし、カスタムメソッドで、Core Dataの永続プロパティに対応するインスタンス変数に直接アクセスしたい、という場合には有用でしょう。以下に、パブリックなアクセサメソッドとプリミティブなアクセサメソッドを対比して示します。`int16`という、`Integer 16`型の属性で、カスタムインスタンス変数`nonCompliantKVCivar`にアクセスするものです。

```
// プリミティブな取得アクセサ
- (short)primitiveInt16 {
    return nonCompliantKVCivar;
}

// プリミティブな設定アクセサ
- (void)setPrimitiveInt16:(short)newInt16 {
    nonCompliantKVCivar = newInt16;
}

// パブリックな取得アクセサ
- (short)int16 {
    short tmpValue;
    [self willAccessValueForKey: @"int16"];
    tmpValue = nonCompliantKVCivar;
    [self didAccessValueForKey: @"int16"];
    return tmpValue;
}

// パブリックな設定アクセサ
- (void)setInt16:(short)int16 {
    [self willChangeValueForKey: @"int16"];
    nonCompliantKVCivar = int16;
    [self didChangeValueForKey:@"int16"];
}
```

管理オブジェクトの生成と削除

Core Dataフレームワークを活用すれば、データを生成する（モデル）オブジェクトの管理に必要となるさまざまな機構を、自分で実装する手間から解放されます。しかしそのためには、モデルオブジェクトを`NSManagedObject`またはその派生クラスのインスタンスとし、Core Dataインフラストラクチャに適切に組み込む必要があります。この文書ではまず、管理オブジェクトを生成するために必要となるインフラストラクチャの基本部分について説明し、管理オブジェクトのインスタンスを生成してインフラストラクチャに組み込む手順を示します。次に、管理オブジェクトを生成するプロセスを説明します。実際には、使いやすいよう、このプロセスはコンビニエンスメソッドとしてまとまっています。さらに、オブジェクトに具体的なストアを割り当てる方法、管理オブジェクトを削除する方法について述べます。

管理オブジェクトの生成、初期化、保存

管理オブジェクトは、Objective-Cのあるクラスのインスタンスです。この点では、ほかのオブジェクトと何の違いもありません。通常通り、`alloc`でインスタンスを生成するだけです。管理オブジェクトがほかのオブジェクトと違うのは、次の3点です。

- `NSManagedObject`またはその派生クラスのインスタンスでなければなりません。
- 管理オブジェクトコンテキストで定義される環境に存在します。
- 「エンティティ記述」が対応づけられており、ここにオブジェクトのプロパティを記述するようになっています。

管理オブジェクトを生成し、Core Dataインフラストラクチャに適切に組み込むためには、本来、さまざまな処理が必要です。しかし実際には、`NSEntityDescription`のクラスメソッドである`insertNewObjectForEntityForName:inManagedObjectContext:`を使って、簡単に実装できるようになっています。「Employee」というエンティティのインスタンスを生成する、最も簡単なコード例を以下に示します。

```
NSManagedObject *newEmployee = [NSEntityDescription
    insertNewObjectForEntityForName:@"Employee"
    inManagedObjectContext:context];
```

このメソッドの戻り値は、管理オブジェクトモデルで定義されたクラスで、エンティティを表現するものです。モデルのエンティティに指定されたデフォルト値を使って初期化されています。

多くの場合、モデルにデフォルト値を指定しておくだけで、初期化には充分です。しかし場合によっては、初期化のために追加の処理が必要になるでしょう。モデルでは表現できない、動的に決まる値（現在の日付や時刻など）で初期化するような状況です。典型的なCocoaアプリケーションでは、当該クラスの`init`メソッドをオーバーライドするのが普通です。しかし`NSManagedObject`の場合、`initWithEntity:insertIntoManagedObjectContext:`をオーバーライドする方法はお勧めしません。Core Dataには、代わりにいくつか、値を初期化する手段が用意されています（「オブジェクトのライフサイクル: 初期化と割り当て解除」（41 ページ）を参照）。

管理オブジェクトを生成しても、それだけでは永続ストアに保存されません。管理オブジェクトコンテキストには、いわゆる「作業領域」としての働きがあります。オブジェクトを生成してここに登録し、変更を施し、その変更を取り消し、再実行することができます。あるコンテキストに関連付けられた管理オブジェクトに変更を施しても、コンテキストに`save:`メッセージを送ってコミットするまでは、当該コンテキストの範囲内にとどまります。コミットした時点で初めて、（検証エラーがなければ）ストアに保存されるのです。

「オブジェクトをストアに割り当て」（57 ページ）も参照してください。

管理オブジェクトの生成過程で内部的に起こっていること

`NSEntityDescription`のコンビニエンスメソッドを使えば管理オブジェクトを簡単に生成、設定できますが、内部的にどのようなことが起こっているか、詳細を知っておけば得るところが大きいでしょう。興味がなければ「オブジェクトをストアに割り当て」（57 ページ）まで飛ばしても構いませんが、機会を設けてこの節を読み返し、このプロセスを充分に把握するようお勧めします。

管理オブジェクトをCore Dataインフラストラクチャに適切に組み込むためには、次の2つが必要です。

- 管理オブジェクトコンテキスト
- エンティティ記述

管理オブジェクトコンテキスト

コンテキストには、管理オブジェクトと、Core Dataインフラストラクチャのその他の機能とを仲介する役割があります。一方、インフラストラクチャの側には、たとえば、管理オブジェクトに対する変更を、コンテキストが管理する「取り消し」アクションに変換したり、管理オブジェクトに割り当てられた永続ストア上で実行するべき処理に変換したりする役割があります。

コンテキストは実質的に、Core Dataのさまざまな機能を利用するための「窓口」として働きます。したがって、コンテキストの参照を保持しておくか、必要なとき直ちにこれを検索できる手段を用意する必要があります。たとえば、`NSPersistentDocument`を用いて文書ベースのアプリケーションを開発しているならば、文書クラスの`managedObjectContext`メソッドを、この手段として利用できます。

エンティティ記述

エンティティ記述には、エンティティの名前、エンティティを表現するクラス、プロパティなどを指定します。エンティティ記述が重要なのは、同じクラスが複数のエンティティを表現することがあるからです。デフォルトでは、エンティティはすべて`NSManagedObject`で表現されます。Core Dataはエンティティ記述を参照して、管理オブジェクトにどのようなプロパティがあるか、どれを永続ストアに保存し、検索できるようにする必要があるか、プロパティ値にどのような制約があるか、などを判断します。エンティティ記述は、管理オブジェクトモデルのプロパティです。

管理オブジェクトコンテキストが与えられれば、次のコード例のように、永続ストアコーディネータを介して適切なエンティティ記述を取得できます。

```
NSManagedObjectContext *context = <#Get a context#>;
NSManagedObjectModel *managedObjectModel =
    [[context persistentStoreCoordinator] managedObjectModel];
NSEntityDescription *employeeEntity =
    [[managedObjectModel entitiesByName] objectForKey:@"Employee"];
```

実際には次の例のように、同じ処理をするNSEntityDescriptionのコンビニエンスメソッドentityForName:inManagedObjectContext:を使うとよいでしょう。

```
NSManagedObjectContext *context = /* これが存在するものと仮定する */;
NSEntityDescription *employeeEntity = [NSEntityDescription
            entityForName:@"Employee"
            inManagedObjectContext:context];
```

管理オブジェクトの生成

NSManagedObjectも基本的に、Objective-Cのクラスであることに変わりはありません。インスタンス生成にはallocを使います。

ほかの多くのクラスと同様、NSManagedObjectのインスタンス生成にはいくつか制約があります。先に説明したように、新たに生成した管理オブジェクトのインスタンスには、そのプロパティを定義するエンティティオブジェクト、環境を定義する管理オブジェクトコンテキストを関連付ける必要があります。したがって、initメッセージを送るだけでは、管理オブジェクトの初期化はできません。専用の初期化メソッドであるinitWithEntity:insertIntoManagedObjectContext:を使って、エンティティとコンテキストを関連付けてください。

```
NSManagedObject *newEmployee = [[NSManagedObject alloc]
            initWithEntity:employeeEntity
            insertIntoManagedObjectContext:context];
```

NSEntityDescriptionのコンビニエンスメソッドであるinsertNewObjectForEntityForName:inManagedObjectContext:は、実際にこのような処理をしています（ただしこのメソッドの戻り値は自動解放されたオブジェクト）。また、「エンティティ記述」（56 ページ）で説明した、エンティティインスタンスの検索処理も行います。したがって通常は、NSManagedObjectのinitWithEntity:insertIntoManagedObjectContext:ではなく、このメソッドを使うようにしてください。

もうひとつ重要なのは、initWithEntity:insertIntoManagedObjectContext:の戻り値が、エンティティ記述で当該エンティティを表現するよう指定されたクラスのインスタンスであることです。Employeeオブジェクトを生成する場合、Employeeエンティティはカスタムクラス（たとえばEmployee）で表現する、とモデルで指定していたとすれば、戻り値はEmployeeのインスタンスになります。一方、EmployeeエンティティをNSManagedObjectで表現することにしていれば、戻り値はNSManagedObjectのインスタンスです。

オブジェクトをストアに割り当て

通常、永続ストアはエンティティごとに1つしかありません。オブジェクトの管理オブジェクトコンテキストを保存する際、Core Dataは自動的に、新しいオブジェクトをこのストアに保存するようになっています。しかし、場合によっては、あるエンティティに対して、書き込み可能なストアを複数用意したいこともあります。たとえば、一部のデータを特定の文書に保存し、残りを共通のグ

ローバルリポジトリ（たとえばユーザの「Application Support」フォルダに置いたストア）に保存する、というような状況です。この場合、オブジェクトをどのストアに置くか指定しなければなりません。

オブジェクトを置くストアは、`NSManagedObjectContext`の`assignObject:toPersistentStore:`メソッドで指定します。その第2引数として、ストアの識別子を指定するようになっています。ストアの識別子は、永続ストアコーディネータから、たとえば`persistentStoreForURL:`で取得できます。管理オブジェクトを生成し、グローバルストアに割り当てるコード例を以下に示します。

```
NSURL *storeURL = <#URL for path to global store#>;

id globalStore = [[context persistentStoreCoordinator]
    persistentStoreForURL:storeURL];

NSManagedObject *newEmployee = [NSEntityDescription
    insertNewObjectForEntityForName:@"Employee"
    inManagedObjectContext:context];

[context assignObject:newEmployee toPersistentStore:globalStore];
```

もちろん、管理オブジェクトコンテキストを保存しない限り、オブジェクトもストアに保存されません。

管理オブジェクトの削除

管理オブジェクトの削除方法に紛らわしいところはありません。その管理オブジェクトコンテキストに、引数として当該オブジェクトを渡して、`deleteObject:`メッセージを送るだけです。

```
[aContext deleteObject:aManagedObject];
```

これで管理オブジェクトはオブジェクトグラフから削除されます。生成したオブジェクトが、コンテキストを保存するまで実際には保存されないのと同様に、削除したオブジェクトも、実際に削除されるのはコンテキストを保存した時点です。

関係

管理オブジェクトを削除する際には、その関係、特に、関係に対して指定されている削除規則について、考慮しておく必要があります。管理オブジェクトの関係に指定された削除規則がすべてNullifyであれば、そのオブジェクトに対してさらに施すべき作業はありません（関係先として指定されていたほかのオブジェクトについては、考慮する必要があるかもしれません。逆方向の関係が必須、あるいは基数の下限が決まっている場合、関係先オブジェクトが不正な状態になる可能性があります）。関係の削除規則がCascadeであれば、あるオブジェクトを削除すると、関係で結ばれた先のオブジェクトも削除されます。Denyであれば、事前に関係先オブジェクトとの関係を削除（切断）しておかなければなりません。これを怠ると、保存の際に検証エラーが発生します。削除規則がNo Actionであれば、オブジェクトグラフの整合性を維持するために必要な処理を、すべて自分で実行しなければなりません。詳しくは「関係の削除規則」（80 ページ）を参照してください。

削除する旨の状態と通知

管理オブジェクトに削除する旨の印がついているかどうかは、`isDeleted`メッセージを送って確認できます。戻り値が`YES`であれば、次に保存する時点で、当該オブジェクトは削除されることになります。つまり、現在の（保留になっている）トランザクションでは、当該オブジェクトには削除対象である旨の印がついている、ということです。さらに、管理オブジェクトコンテキストに`deleteObject:`メッセージを送ると、コンテキストは、新たに削除されたオブジェクトを「削除されたオブジェクト」リストに入れて、`NSManagedObjectContextObjectsDidChangeNotification`という通知をポストします。なお、コンテキストから削除する旨の印がついたオブジェクトと、永続ストアから削除する旨の印がついたオブジェクトは、同じではないので注意が必要です。オブジェクトを生成し、次いで削除する操作を、同じトランザクション内で（すなわち、途中で保存操作をすることなしに）実行した場合、`NSManagedObjectContext`の`deletedObjects`メソッドから返される配列、あるいは、`NSManagedObjectContextDidSaveNotification`通知の「削除されたオブジェクト」リストに、当該オブジェクトは現れません。

管理オブジェクトのフェッチ

この章では、管理オブジェクトをフェッチする手順を示し、効率的にフェッチするための考慮事項について説明します。また、NSExpressionDescriptionオブジェクトを使って特定の値を検索する方法も解説します。さまざまな状況で使えるコード例が『*Core Data Snippets*』に載っているので参考にしてください。

管理オブジェクトのフェッチ

管理オブジェクトのフェッチは、フェッチ要求を管理オブジェクトコンテキストに送信することにより行います。したがって、まずフェッチ要求を作成しなければなりません。ここには最低限、要求するエンティティを指定する必要があります。管理オブジェクトモデルからエンティティを取得するためには、NSEntityDescriptionのentityForName:inManagedObjectContext:メソッドを使います。必要ならば、述語（述語の作成については『*Predicate Programming Guide*』を参照）、整列記述子その他の属性も指定できます。次に、以下の例のように、executeFetchRequest:error:を使って、コンテキストからオブジェクトを検索します。

リスト 1　　フェッチ要求を作成、実行するコード例

```
NSManagedObjectContext *moc = [self managedObjectContext];
NSEntityDescription *entityDescription = [NSEntityDescription
    entityForName:@"Employee" inManagedObjectContext:moc];
NSFetchRequest *request = [[[NSFetchRequest alloc] init] autorelease];
[request setEntity:entityDescription];

// 述語および整列順序を設定...
NSNumber *minimumSalary = ...;
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"(lastName LIKE[c] 'Worsley') AND (salary > %@)", minimumSalary];
[request setPredicate:predicate];

NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
    initWithKey:@"firstName" ascending:YES];
[request setSortDescriptors:[NSArray arrayWithObject:sortDescriptor]];
[sortDescriptor release];

NSError *error = nil;
NSArray *array = [moc executeFetchRequest:request error:&error];
if (array == nil)
{
    // エラー処理...
}
```

一時プロパティに基づく述語を使ってフェッチすることはできません（メモリ中でしぼり込みに使うことは可能です）。ほかにもいくつか、フェッチ処理とストアの型に関して考慮するべき事項があります詳しくは「ストアの型と動作」（125 ページ）で説明しますが、あらかじめ簡単に述べておくと、フェッチ処理を直接実行する場合、一般に、*Objective-C*ベースの述語や整列記述子をフェッ

チ要求に入れない方がよい、ということです。まとめてフェッチした後、その結果をしぼり込み、整列してください。配列コントローラを使う場合、整列記述子を永続ストアに渡す代わりに、NSArrayControllerのサブクラスを定義して、データをフェッチした後で整列する、という形にする必要があるかもしれません。

アプリケーションで複数の永続スタックを使っている、または複数のアプリケーションが同時に同じストアにアクセス（し、修正）する場合、フェッチ処理を行うことにより、データの値が最新であることを保証できます（「データが最新であることの保証」（72 ページ）を参照）。

特定のオブジェクトの取得

アプリケーションで複数のコンテキストを使っており、オブジェクトが永続ストアから削除されたかどうか調べたい場合は、「self == %@」という形式の述語を指定したフェッチ要求を作成してください。変数として渡すオブジェクトは、次の例のように、管理オブジェクトでも管理オブジェクトIDでも構いません。

```
NSFetchRequest *request = [[[NSFetchRequest alloc] init] autorelease];
NSEntityDescription *entity =
    [NSEntityDescription entityForName:@"Employee"
            inManagedObjectContext:managedObjectContext];
[request setEntity:entity];

NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"self == %@", targetObject];
[request setPredicate:predicate];

NSError *error = nil;
NSArray *array = [managedObjectContext executeFetchRequest:request error:&error];
if (array != nil) {
    NSUInteger count = [array count]; // オブジェクトが削除されていれば0。
    //
}
else {
    // エラー処理。
}
```

該当するオブジェクトが削除されていれば、フェッチ処理の戻り値である配列の要素数は0になります。複数のオブジェクトについて、現存するかどうか調べたい場合は、個別にフェッチ処理を行うよりも、次のように、IN演算子を使う方が効率的です。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"self IN %@",
                                        arrayOfManagedObjectIDs];
```

特殊な値のフェッチ

管理オブジェクトを実際にフェッチすることなく、たとえばある属性の最大値や最小値を調べたい、ということもあるでしょう。Mac OS X v10.6以降およびiOSでは、NSExpressionDescriptionを使って、条件に合致する値を直接検索できます。

通常のフェッチ操作と同じように、フェッチ要求オブジェクトを生成してエンティティを設定するのですが、次の点が異なります。

- 戻り値として辞書を返すよう指定してください。

  フェッチ要求に対し、引数としてNSDictionaryResultTypeを指定して、setResultType:メッセージを送ります。

- NSExpressionDescriptionのインスタンスを生成し、検索するプロパティを指定してください。

  取得したい値が1つ（たとえばEmployeeテーブルの最高給与額）ならば、式記述を1つだけ生成します。

式記述を生成、使用する手順を以下に示します。

1. まず、検索したい値のキーパス、適用する関数（max:、min:など）を表す式（NSExpressionのインスタンス）を生成します。

   ```
   NSExpression *keyPathExpression = [NSExpression expressionForKeyPath:@"salary"];
   NSExpression *maxSalaryExpression = [NSExpression expressionForFunction:@"max:"
                                                                 arguments:[NSArray
   arrayWithObject:keyPathExpression]];
   ```

   ここに指定できる関数の完全なリストは、expressionForFunction:arguments:を参照してください。

2. 式記述を生成し、その名前、式、戻り値型を設定します。

   名前は辞書に戻り値を格納する際にキーとして使われる。複数の値（たとえばEmployeeテーブルの最高給与額と最低給与額）を検索したい場合、式記述の名前は、当該フェッチ要求の範囲で一意的でなければなりません。

   ```
   NSExpressionDescription *expressionDescription = [[NSExpressionDescription alloc]
    init];
   [expressionDescription setName:@"maxSalary"];
   [expressionDescription setExpression:maxSalaryExpression];
   [expressionDescription setExpressionResultType:NSDecimalAttributeType];
   ```

3. 最後に、要求のプロパティとして、式が表すプロパティのみフェッチする旨を設定します。

   ```
   [request setPropertiesToFetch:[NSArray arrayWithObject:expressionDescription]];
   ```

すると、通常と同じように、executeFetchRequest:error:を使ってフェッチ要求を実行できます。戻り値は辞書の内容が収容された配列です。式記述の名前をキーとして、要求した値が格納されています。

「Event」というエンティティについて、属性「creationDate」の最小値を取得する例を示します。

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Event"
inManagedObjectContext:context];
[request setEntity:entity];

// 戻り値は辞書である旨を指定。
[request setResultType:NSDictionaryResultType];
```

```
// キーパスの式を生成。
NSExpression *keyPathExpression = [NSExpression
expressionForKeyPath:@"creationDate"];

// キーパス「creationDate」の最小値を表す式を生成。
NSExpression *minExpression = [NSExpression expressionForFunction:@"min:"
arguments:[NSArray arrayWithObject:keyPathExpression]];

// minExpressionを使って日付を返す式記述を生成。
NSExpressionDescription *expressionDescription = [[NSExpressionDescription alloc]
 init];

// 名前は辞書に戻り値を格納する際にキーとして使われる。
[expressionDescription setName:@"minDate"];
[expressionDescription setExpression:minExpression];
[expressionDescription setExpressionResultType:NSDateAttributeType];

// 要求のプロパティとして、式で表されるプロパティのみフェッチする旨を指定。
[request setPropertiesToFetch:[NSArray arrayWithObject:expressionDescription]];

// フェッチ処理の実行。
NSError *error = nil;
NSArray *objects = [managedObjectContext executeFetchRequest:request
error:&error];
if (objects == nil) {
    // エラーを処理する。
}
else {
    if ([objects count] > 0) {
        NSLog(@"Minimum date: %@", [[objects objectAtIndex:0]
valueForKey:@"minDate"]);
    }
}

[expressionDescription release];
[request release];
```

フェッチ処理とエンティティの継承

エンティティの継承階層（「エンティティの継承」（29 ページ）を参照）を定義している場合、フェッチ要求のエンティティとしてスーパーエンティティを指定すれば、スーパーエンティティおよびサブエンティティの、条件に合致するインスタンスすべてをフェッチできます。アプリケーションによっては、スーパーエンティティを抽象エンティティと指定しているかもしれません（「抽象エンティティ」（30 ページ）を参照）。ある抽象エンティティの子である具象サブエンティティすべてを対象として、条件に合致するインスタンスをフェッチしたい場合は、フェッチ仕様のエンティティとして、当該抽象エンティティを指定します。「抽象エンティティ」に挙げて例の場合、Graphicエンティティを指定してフェッチ要求すると、Circle、TextArea、Lineのインスタンスを取得できます。

管理オブジェクトの使い方

この文書では、アプリケーションで管理オブジェクトを利用、操作する際のさまざまな問題点を解説します。

プロパティのアクセスと修正

管理オブジェクトクラスについては、モデル化プロパティ（属性および関係）の取得/設定アクセサメソッドが自動的に生成され、パブリックおよびプリミティブなメソッドとして効率的に使えます（「管理オブジェクトのアクセサメソッド」（45 ページ）を参照）。管理オブジェクトのプロパティにアクセスし、修正するためには、このメソッドを直接使わなければなりません。

関係の多くは、元来、両方向です。オブジェクト間の関係に対する変更は、オブジェクトグラフの整合性を維持しながら施さなければなりません。ある方向とその逆方向の関係を正しくモデル化し、関係を設定する際、逆方向の関係もきちんと設定していれば、関係で結ばれた一方のオブジェクトを修正したとき、もう一方も自動的に更新されます（「関係の操作とオブジェクトグラフの整合性」（81 ページ）を参照）。

属性および対1の関係

管理オブジェクトの属性や対1関係には、次のコード例のように、標準的なアクセサメソッド、またはObjective-C 2.0のドット構文（「ドット構文」 in *The Objective-C Programming Language*を参照）でアクセスします。

```
NSString *firstName = [anEmployee firstName];
Employee *manager = anEmployee.manager;
```

属性を修正する場合も同様に、標準的なアクセサメソッドまたはドット構文を使います（次のコード例を参照）。

```
newEmployee.firstName = @"Stig";
[newEmployee setManager:manager];
```

取得、設定とも、ドット構文は、標準的なメソッドの呼び出しとまったく同じ動作です。たとえば次の2つの文は、同等なコードパスを使っています。

```
[[aDepartment manager] setSalary:[NSNumber numberWithInteger:100000]];
aDepartment.manager.salary = [NSNumber numberWithInteger:100000];
```

注： カスタムアクセサを使おうとしたとき、コンパイラが警告またはエラーを出す場合は、該当するプロパティを宣言するNSManagedObjectのカテゴリを宣言するか、または当該プロパティを宣言しているエンティティに対応する、NSManagedObjectのカスタムサブクラスを実装してください（通常は後者の方法を推奨、「管理オブジェクトのアクセサメソッド」（45 ページ）を参照）。

単純な属性値の取得/設定には、キー値コーディング（KVC）も使えます（次のコード例を参照）。ただし、アクセサメソッドを使う方法に比べて効率は劣るので、どうしても必要な場合（キーやキーパスを動的に選択するなど）に限って使うとよいでしょう。

```
[newEmployee setValue:@"Stig" forKey:@"firstName"];
[aDepartment setValue:[NSNumber numberWithInteger:100000]
forKeyPath:@"manager.salary"];
```

ただし、属性値の変更は、KVCに準拠した方式でなければなりません。たとえば次のように記述すると、プログラミングエラーになります。

```
NSMutableString *mutableString = [NSMutableString stringWithString:@"Stig"];
[newEmployee setFirstName:mutableString];
[mutableString setString:@"Laura"];
```

ミュータブルな値については、値の所有権をCore Dataに渡すか、常にコピーを実行するカスタムアクセサメソッドを実装する必要があります。先の例で、Employeeエンティティを表現するクラスにfirstNameプロパティ（copy）が宣言されていれば（あるいは新しい値をコピーするカスタムsetFirstName:メソッドが実装されていれば）、エラーにはなりません。この場合、setString:の呼び出し（コード例の3行目）後も、firstNameの値は「Stig」のままであって、「Laura」にはなりません。

カスタムアクセサメソッド内を除き、プリミティブなアクセサメソッドを呼び出さなければならないことは、ほとんどないでしょう（「管理オブジェクトのアクセサメソッド」（45 ページ）を参照）。

対多の関係

対多関係（1対多の関係先オブジェクト、多対多の関係で結ばれた双方のオブジェクト）にアクセスするためには、標準的な取得アクセサメソッドを使います。対多の関係は、次のコード例のように、集合の形で表現されます。

```
NSSet *managersPeers = [managersManager directReports];
NSSet *departmentsEmployees = aDepartment.employees;
```

関係先にアクセスした時点で最初に受け取るのは、「フォールト」状態のオブジェクトかもしれません（「フォールティングと一意化」（105 ページ）を参照）。これに何らかの変更を施すと、自動的にフォールトが「発動」します（通常、関係がフォールト状態かどうかを意識する必要はありませんが、NSManagedObjectのhasFaultForRelationshipNamed:メソッドで調べることは可能です）。

原理的には対多関係もすべて、対1関係と同じように、カスタムアクセサメソッドまたはキー値コーディング（こちらの方が望ましい）を使って、次の例のように操作できます。

```
NSSet *newEmployees = [NSSet setWithObjects:employee1, employee2, nil];
[aDepartment setEmployees:newEmployees];

NSSet *newDirectReports = [NSSet setWithObjects:employee3, employee4, nil];
```

```
manager.directReports = newDirectReports;
```

しかし通常は、関係を一括して設定するよりも、要素をひとつずつ追加/削除する方が扱いやすいでしょう。それには、mutableSetValueForKey:、または自動生成される適当な関係ミューテータメソッドを使います（「動的に生成されるアクセサメソッド」（46 ページ）を参照）。

```
NSMutableSet *employees = [aDepartment mutableSetValueForKey:@"employees"];
[employees addObject:newEmployee];
[employees removeObject:firedEmployee];

// または
[aDepartment addEmployeesObject:newEmployee];
[aDepartment removeEmployeesObject:firedEmployee];
```

ドット構文によるアクセサの戻り値と、mutableSetValueForKey:の戻り値の違いを理解しておくことが大切です。mutableSetValueForKey:の戻り値は、ミュータブルなプロキシオブジェクトです。その内容を変更すると、関係のキー値監視（KVO）変更通知が適切に送出されます。一方、ドット構文のアクセサは、単に集合を返すだけです。集合を次のコード例のように操作しても:

```
[aDepartment.employees addObject:newEmployee]; // 不適切なコード!
```

KVO変更通知は送出*されず*、逆方向の関係が適切に更新されることもありません。

先に述べたようにドット構文は単にアクセサメソッドを起動するだけなので、同じ理由で、次のようなコードも不適切です。

```
[[aDepartment employees] addObject:newEmployee]; // 不適切なコード!
```

変更内容の保存

管理オブジェクトに変更を施しても、それだけではストアに反映されません。管理オブジェクトコンテキストには、いわゆる「作業領域」としての働きがあります。管理オブジェクトを生成してここに登録し、変更を施し、その変更を取り消し、再実行することができます。あるコンテキストに関連付けられた管理オブジェクトに変更を施しても、コンテキストにsave:メッセージを送ってコミットするまでは、当該コンテキストの範囲内にとどまります。コミットした時点で初めて、（検証エラーがなければ）ストアに保存されるのです。同様に、管理オブジェクトを生成するだけでは永続ストアに保存されず、逆に管理オブジェクトを削除してもそれだけでレコードが削除されることはありません。コンテキストを保存して、変更内容をコミットする必要があります。

「データが最新であることの保証」（72 ページ）も参照してください。

管理オブジェクトのIDとURI

NSManagedObjectIDオブジェクトは管理オブジェクトの汎用的な識別子で、Core Dataフレームワークにおいては、一意性を確保するための基盤として働きます。管理オブジェクトIDを使えば、同一アプリケーション内だけでなく、（たとえば分散システムにおける）複数のアプリケーションにまたがる管理オブジェクトコンテキスト間で、同じ管理オブジェクトを一意に識別できます。データベースの主キーと同様、識別子には、永続ストア内のオブジェクトを厳密に特定するために必要な

情報が含まれています。もっとも、具体的な実装詳細を表に出すことはありません。フレームワークは、ほかのオブジェクトが知る必要のない情報を完全にカプセル化（隠蔽）し、純粋なオブジェクト指向インターフェイスのみを公開します。

```
NSManagedObjectID *moID = [managedObject objectID];
```

オブジェクトIDには2つの形式があります。管理オブジェクトを生成した時点では一時IDを割り当て、永続ストアに保存する際に初めて、永続IDを割り当てるようになっています。あるIDが一時IDか否かは、次のようにして調べることができます。

```
BOOL isTemporary = [[managedObject objectID] isTemporaryID];
```

また、オブジェクトIDをURI表現に変換できます。

```
NSURL *moURI = [[managedObject objectID] URIRepresentation];
```

管理オブジェクトIDまたはそのURI表現が与えられれば、該当する管理オブジェクトは、`managedObjectIDForURIRepresentation:`または`objectWithID:`で検索できます。

URI表現には、アーカイブできるという利点があります。これに対し、一時IDは将来変わってしまうものなので、一般に、アーカイブするべきではありません。たとえば、アーカイブしたURIを、ユーザごとの環境設定ファイルに格納する、という方法で、テーブルビューで最近選択したオブジェクトグループを保存しておくようなことが可能です。また、URIを使えば、コピーとペーストの操作（「コピー処理、コピーとペースト」（68 ページ）を参照）、ドラッグ&ドロップ操作（「ドラッグ&ドロップ」（69 ページ）を参照）を実装することも可能です。

オブジェクトIDを使って、（ハードジョインができない）永続ストアをまたがる、「弱い」関係を定義できます。たとえば弱い対多関係については、アーカイブしたURIとして、関係で結ばれた先のオブジェクト群のIDを保存し、オブジェクトIDをもとに導出した一時属性として関係を管理するのです。

場合によっては、独自の一意ID（UUID）プロパティを生成すれば役に立つかもしれません。これは新たに挿入したオブジェクトに対して定義、設定できます。こうしておけば、述語を使って効率的にオブジェクトを検索できます（保存操作の前でも、新たに生成したオブジェクトを検索可。ただし、生成した元のコンテキスト内でのみ）。

コピー処理、コピーとペースト

管理オブジェクトを対象として、汎用的な形でコピー処理にまつわる問題を解決し、コピーとペーストの機能を提供するのは困難です。管理オブジェクトのどのプロパティを実際にコピーするべきなのか、状況に応じて判断しなければなりません。

属性のコピー処理

管理オブジェクトの属性をコピーしたいだけならば、多くの場合、最善の戦略は、（1）コピー操作では管理オブジェクトの辞書（プロパティリスト）表現を生成し、（2）ペースト操作の際には、新たに管理オブジェクトを生成し、この辞書を使ってプロパティ値を複製する、というものです。管理オブジェクトのID（「管理オブジェクトのIDとURI」（67 ページ）を参照）を使って、コピー/ペースト処理を実装できます。ただし、そのためには、新たに生成したばかりのオブジェクトも、コピーが可能であるための条件に適合させなければなりません。

まだ保存していない、新しい管理オブジェクトには、一時IDが割り当てられています。コピー操作後に保存操作をすると、管理オブジェクトのIDが変わり、コピーの際に記録したIDは無効になるため、ペースト操作ができなくなってしまいます。これを回避するため、「遅延書き込み」を行います（「コピーとペースト」を参照）。コピー操作では、カスタム型を宣言しますが、管理オブジェクトのIDが一時IDであれば、実際にはデータを書き込まず、元の管理オブジェクトの参照を保持しておきます。次に、`pasteboard:provideDataForType:`メソッドで、その時点におけるオブジェクトのIDを書き込みます。

さらに面倒なことに、まだ一時IDである状態でペースト操作をしたとしても、その後、保存した後で再びペーストする、という可能性を考えておかなければなりません。したがって、ペーストボード上で型を再宣言して、改めて遅延ペーストの設定をする必要があります。こうしておかないと、ペーストボードには一時IDが保持されたままになります。`pasteboard:provideDataForType:`の実行中に`addTypes:owner:`を呼び出すことはできないので、次のように、遅延実行しなければなりません。

```
- (void)pasteboard:(NSPasteboard *)sender provideDataForType:(NSString *)type
{
    if ([type isEqualToString:MyMOIDType]) {
        // cachedManagedObjectは当初コピーされたオブジェクトであると想定
        NSManagedObjectID *moID = [cachedManagedObject objectID];
        NSURL *moURI = [moID URIRepresentation];
        [sender setString:[moURI absoluteString] forType:MyMOIDType];
        if ([moID isTemporaryID]) {
            [self performSelector:@selector(clearMOIDInPasteboard:)
                    withObject:sender afterDelay:0];
        }
    }
    // 実装が続く...
}

- (void)clearMOIDInPasteboard:(NSPasteboard *)pb
{
    [pb addTypes:[NSArray arrayWithObject:MyMOIDType] owner:self];
}
```

関係のコピー

関係をコピーしたいのであれば、関係で結ばれた先のオブジェクトについても考慮しておく必要があります。その処理を誤ると、オブジェクトグラフ全体をまるごとコピーすることになりかねません（これが期待する動作ではないはずです）。対1関係をコピーする際には、関係先オブジェクトのコピーを新たに生成するか、参照で済ませるかを考慮する必要があります。参照で済ませた場合、関係元のオブジェクトに戻る、逆向きの関係はどうなるでしょうか。コピーするためには、ほかのオブジェクトとの関係を再定義するべきでしょうか。対多関係についても同様の考慮が必要です。

ドラッグ&ドロップ

URI表現を使って、管理オブジェクトに対するドラッグ&ドロップ操作ができます（「管理オブジェクトのIDとURI」（67 ページ）を参照）。オブジェクトをある関係から別の関係につなぎ替える場合などに便利でしょう。

```
NSURL *moURI = [[managedObject objectID] URIRepresentation];
```

ドラッグ処理用のペーストボードにURIを記録しておき、後でこれを検索して、元の管理オブジェクトの参照を、永続ストアコーディネータを使って生成し直します（次のコード例を参照）。

```
NSURL *moURL = // ペーストボードから取得...
NSManagedObjectID *moID = [[managedObjectContext persistentStoreCoordinator]
    managedObjectIDForURIRepresentation:moURL];
// moIDはnilでないと想定...
NSManagedObject *mo = [managedObjectContext objectWithID:moID];
```

ここでは、ドラッグ&ドロップが「同じ永続スタックの範囲内」である（すなわち、複数の管理オブジェクトコンテキストが関与していれば、共有永続ストアコーディネータを使っている）か、またはドラッグ&ドロップされるオブジェクトが、永続ストアコーディネータに参照されるストア内にある、と想定しています。

ドラッグ&ドロップ操作によりコピー/ペーストをする場合には、管理オブジェクトを適切な表現にしてペーストボードに記録すること、ドロップメソッドではこの表現を取得し、新たに管理オブジェクトを生成することが必要です（「コピー処理、コピーとペースト」（68ページ）を参照）。

検証

Core Dataフレームワークには、検証を支援する、分かりやすい構造の基盤が備わっています。オブジェクトモデルにカプセル化されたロジックを用いる方式、カスタムコードを用いる方式の両方に対応しています。管理オブジェクトモデルでは、値に対して、正常であれば満たしているべき制約を指定できます（Employeeの給与額は負にならない、どの従業員もいずれかのDepartmentに属している、など）。カスタム検証メソッドには2つの形態があります。標準的なキー値コーディングの規約（「キー値検証」を参照）に基づき、ある単一の属性の値を検証するものと、ライフサイクルの各段階（挿入、更新、削除）でオブジェクト全体を検証する特別なメソッド群（`validateForInsert:`、`validateForUpdate:`、`validateForDelete:`）を用いるものです。後者は、値の組み合わせを検証できる、という点で特に有用です。たとえば、従業員が持ち株制度に加入できるのは、所定の年数以上勤めており、かつ所定の給与等級以上である者に限る、というような条件です。

変更事項を外部ストアにコミットする際には、自動的にモデルベースの制約を検査し、検証メソッドを起動することにより、無効なデータが格納されることのないようにしています。また、必要ならばプログラム上で明示的に起動することも可能です。個々の値は`validateValue:forKey:error:`で検証します。管理オブジェクトは、新しい値をモデルで指定された制約と照合し、カスタム検証メソッド（`validate<Key>:error:`という形の名前のメソッド）を起動するようになっています。したがって、カスタム検証メソッドを実装した場合でも、これを明示的に呼び出す必要はありません。これにより、管理オブジェクトモデルで定義された制約が、きちんと適用されます。

検証メソッドの実装については「モデルオブジェクト検証」を参照してください。

取り消し管理

Core Dataフレームワークには、取り消し/再実行の処理を実装するための支援機能があります。取り消し管理は一時プロパティ（永続ストアに格納されていないけれども、管理オブジェクトモデルに指定されているプロパティ）にも及びます。

管理オブジェクトは管理オブジェクトコンテキストに関連付けられています。そして、管理オブジェクトコンテキストがそれぞれ、取り消しマネージャを保持します。コンテキストはキー値監視の仕組みを使って、登録オブジェクトに対して施された修正を追跡します。管理オブジェクトのプロパティに対する変更には、通常のアクセサメソッド、キー値コーディングのほか、キー値監視の仕組みに準拠してカスタムクラスに定義したカスタムメソッドも使えます。コンテキストは適切なイベントを取り消しマネージャに登録します。

ある操作を取り消す/再実行するためには、単にコンテキストに`undo`メッセージ/`redo`メッセージを送るだけで済んでしまいます。また、最後に保存して以降の変更をすべて取り消す（ロールバック）ことも、`rollback`を呼び出すだけで可能です（取り消しスタックもクリアされます）。コンテキストを初期状態にリセットする`reset`もあります。

さらに、取り消しイベントのグループ化など、取り消しマネージャの標準的な機能も自由に利用できます。グループ化においては、一連の取り消し登録をいったんキューに並べ、取り消し処理をまとめて行います。キューに並べた段階で、まとめて実行しても同じ結果になる事項を併合し、相反する変更を相殺するなどの操作ができるので、単に順序通りに実行するよりも効率が改善されます。`beginUndoGrouping`および`endUndoGrouping`以外のメソッドを使う場合は、キューに入っている操作を適切に一括処理できるよう、最初に管理オブジェクトコンテキストに`processPendingChanges`メッセージを送る必要があります。

状況によっては、一時的に取り消しの動作を変更し、あるいは無効にする必要があるかもしれません。新規文書を作成する際にデフォルトのオブジェクトセットを生成する（しかし、修正を施した旨の表示は出ないようにしたい）、あるいは、ほかのスレッドやプロセスから新しい状態をマージする必要がある、といった状況です。一般に、取り消し登録（取り消し指示に応じられるよう、必要事項を登録しておくこと）なしである操作を実行したい場合は、取り消しマネージャに`disableUndoRegistration`メッセージを送ります。当該操作の終了後、`enableUndoRegistration`メッセージを送ってください。それぞれの前に、コンテキストに`processPendingChanges`メッセージを送ります（次のコード例を参照）。

```
NSManagedObjectContext *moc = ...;
[moc processPendingChanges];  // 取り消しの対象とする操作を一括処理
[[moc undoManager] disableUndoRegistration];
// 取り消し指示に備えて操作を記録することなしに変更を行う
[moc processPendingChanges];  // 取り消しの対象としたくない操作を一括処理
[[moc undoManager] enableUndoRegistration];
```

フォールト

管理オブジェクトは、通常、永続ストアに格納されている、あるデータを表現しています。管理オブジェクトは、「フォールト」、すなわち、プロパティ値が外部ストアから読み込まれていない状態になることがあります。この管理オブジェクトの永続プロパティ値にアクセスすると、フォールトが「発動」し、自動的にストアから値が検索されるようになっています。状況によっては、管理オブジェクトを強制的にフォールト状態にすることがあります（確実に最新の値にするため、`NSManagedObjectContext`の`refreshObject:mergeChanges:`で実行）。特に頻繁にフォールト状態のオブジェクトが見つかるのは、関係をたどる操作をしているときでしょう。

管理オブジェクトをフェッチしても、関係で結ばれたほかのオブジェクトのデータが、自動的にフェッチされるわけではありません（「フォールティングによるオブジェクトグラフの大きさ制限」（105 ページ）を参照）。当初、あるオブジェクトの関係先オブジェクトは、（すでにフェッチ済みである場合を除き）フォールト状態になっています（「同一コンテキスト、同一レコードに対応する管理オブジェクトの一意化」（107 ページ）を参照）。関係先のオブジェクトにアクセスした時点で、自動的にデータが検索されます。たとえばアプリケーション起動時に、Employeeのオ

ブジェクトを1つ、永続ストアからフェッチしたとしましょう。この時点では、関係先であるmanagerとdepartmentは、（永続ストアには存在するとしても）「フォールト」として表現されています。にもかかわらず、当該従業員の上長の名前を、次のようなコードで検索できます。

```
NSString *managersName =
        [[anEmployee valueForKey:@"manager"] valueForKey:@"lastName];
```

あるいはキーパスを用いて次のように記述する方が簡単かもしれません。

```
NSString *managersName =
        [anEmployee valueForKeyPath:@"manager.lastName"];
```

この場合、関係で結ばれた先のEmployeeオブジェクト（すなわち上長）のデータは、自動的に検索されます。

ここに巧妙かつ重要なポイントがあります。関係をたどる（この例では従業員の上長を見つける）際、明示的に関係先オブジェクトをフェッチする（すなわち、フェッチ要求を生成し、実行する）必要はありません。単にキー値コーディングにより（あるいはアクセサメソッドを実装していればそれを使って）関係先オブジェクトを参照するだけで、フェッチ要求は自動的に生成されるのです。たとえば、ある従業員の上長の上長が属する部署の名前は、次のようにして調べることができます。

```
NSString *departmentName = [anEmployee
valueForKeyPath:@"manager.manager.department.name"];
```

ここではもちろん、従業員が指揮系統の3階層目以下であると想定しています。また、コレクション演算子メソッドも使えます。ある従業員が属する部署の総給与額は、次のようにして調べることができます。

```
NSNumber *salaryOverhead = [anEmployee
valueForKeyPath:@"department.employees.@sum.salary"];
```

多くの場合、当初のフェッチでは、オブジェクトグラフの起点ノードのみを取得します。それ以降は、フェッチ要求を実行する代わりに、関係をたどるだけで済みます。

データが最新であることの保証

2つのアプリケーションが同じデータストアを使っていたり、単一のアプリケーションであっても複数の永続スタックがあったりすると、管理オブジェクトコンテキストまたは永続オブジェクトストア内の管理オブジェクトが、リポジトリの内容と一致しなくなる可能性があります。その場合、管理オブジェクト、特に永続オブジェクトストア（スナップショット）のデータを「リフレッシュ」して、データを最新の状態にしなければなりません。

オブジェクトのリフレッシュ

実体化した（プロパティ値を永続ストアから取得した）管理オブジェクトや、更新、挿入、削除が保留になっているオブジェクトは、開発者が何らかの処理を組み込んでいない限り、フェッチ操作により更新されることはありません。たとえば、一方の編集コンテキストで、あるオブジェクトをフェッチし、修正を施したとしましょう。その間に、別の編集コンテキストで同じデータを編集

し、その変更をコミットしたとします。この状態で、1つ目の編集コンテキストで改めてフェッチを実行しても、同じオブジェクトが得られるだけで、コミットされた新しいデータ値にはなりません。その時点でメモリ上にある、既存のオブジェクトしか見えないのです。

管理オブジェクトのプロパティ値をリフレッシュするためには、管理オブジェクトコンテキストの`refreshObject:mergeChanges:`メソッドを使います。このメソッドは、`mergeChanges`フラグが`YES`であれば、オブジェクトのプロパティ値に、永続ストアコーディネータが参照するオブジェクトのプロパティ値をマージします。`NO`であれば、オブジェクトを「フォールト」状態に戻すだけで、値をマージすることはありません（その結果、ほかの関係する管理オブジェクトも解放されるので、オブジェクトグラフのうちメモリ中に保持したい部分のみを残して削除するためにも利用できます）。

なお、オブジェクトが最新の状態でないのは、ストアがスナップショットを再フェッチするまでの期間です。それまでの間にフォールトが発動しなければ、何も起こりません。さらに、これはストアがSQLiteの場合にのみ関係します（ほかのストアは、データ全体がメモリ中に保持されているので、再フェッチは起こりません）。

一時プロパティに対する変更のマージ

`mergeChanges`フラグを`YES`として`refreshObject:mergeChanges:`を実行すると、`awakeFromFetch`を起動したとき、一時プロパティがリフレッシュ前の値に戻るようになります。したがって、ある一時プロパティの値が別のプロパティに依存していて、これがリフレッシュされている場合、一時値の同期が崩れることがあります。

あるアプリケーションにPersonというエンティティがあって、その属性として、`firstName`および`lastName`のほか、これを元に値が決まる*キャッシュされた*一時プロパティ`fullName`があるとします（実際に`fullName`属性をキャッシュすることは考えにくいのですが、ここでは分かりやすくするためこの例を使います）。さらに、`awakeFromFetch`というカスタムメソッドで`fullName`を計算し、キャッシュするとしましょう。

永続ストアで現在「Sarit Smith」と名付けられているPersonを、2つの管理オブジェクトコンテキストで編集する、という状況を考えます。

- コンテキスト1で、対応するインスタンスの`firstName`を「Fiona」と変更し（したがってキャッシュされた`fullName`の値は「Fiona Smith」となる）、コンテキストを保存しました。

  永続ストアでは、この人物は「Fiona Smith」となっています。

- コンテキスト2で、対応するインスタンスの`lastName`を「Jones」と変更し、その結果、キャッシュされた`fullName`の値は「Sarit Jones」となりました。

  この状態で、`mergeChanges`フラグを`YES`として、このオブジェクトをリフレッシュします。すると、ストアから「Fiona Smith」という値が得られます。

  - `firstName`はリフレッシュ前に変更*していません*。リフレッシュにより、永続ストアからの新しい値に更新され、「Fiona」となりました。
  - `lastName`はリフレッシュ前に変更*しています*。したがって、リフレッシュの結果、このコンテキストで修正した値、すなわち「Jones」になります。
  - *一時*値である`fullName`も、リフレッシュ前に変更しています。そのため、リフレッシュの結果、値は「Sarit Jones」になりました（本来は「Fiona Jones」であるべきです）。

この例が示すように、リフレッシュ前の値が`awakeFromFetch`の実行後に適用されるため、リフレッシュ処理により一時値が適切に更新されるとは保証できません（リフレッシュ後に上書きされてしまう）。このような場合、最もよい対処法は、リフレッシュが起こった旨を表すインスタンス変数を追加し、必要に応じて一時値を計算し直すことです。たとえば**Person**クラスの場合、`BOOL`型のインスタンス変数`fullNameIsValid`を宣言し、この値を`NO`とする`didTurnIntoFault`メソッドを実装します。次に、`fullName`属性のカスタムアクセサを実装します。ここでは、`fullNameIsValid`の値を調べ、`NO`であれば再計算します。

Core Dataによるメモリ管理

一般に、メモリ管理については、Core Dataを使う場合でも、Cocoaに適用されていた従来通りのガイドラインに従ってください。ただし、ほかにいくつか考慮しなければならない事項があります。

注： Mac OS X v10.5以降、ガベージコレクションが稼働する環境でもCore Dataが使えるようになりました（『*Garbage Collection Programming Guide*』を参照）。この章の解説のうち、もっぱら参照カウント環境に関係するものは、ガベージコレクションを採用するならば適用されません（たとえば循環参照が問題になることはありません。「関係の循環参照の切断」（76 ページ）を参照）。

インスタンスとデータのライフサイクル

重要なのは、一般に、管理オブジェクトが表すデータのライフサイクルが、管理オブジェクトの個々のインスタンスのそれには依存しない、ということです。レコードを永続ストアに追加するためには、管理オブジェクト用の領域を割り当て、初期化した上で、管理オブジェクトコンテキストを保存する必要があります。レコードを永続ストアから削除した場合は、ある時点で、対応する管理オブジェクトの割り当てを解除しなければなりません。しかし、追加から削除までの間に、ある永続ストアの同じレコードを表す管理オブジェクトのインスタンスを、いくつでも生成、破棄できます。

`NSEntityDescription`のコンビニエンスメソッド`insertNewObjectForEntityForName:inManagedObjectContext:`を使えば、管理オブジェクトを生成し、編集中のコンテキストに挿入する処理を一括して実行できます。メソッド名の先頭が「new」*ではない*ので、参照カウント環境では、返されたオブジェクトは呼び出し元の所有になりません（「メモリ管理の規則」を参照）。

管理オブジェクトコンテキストの役割

管理オブジェクトは、自身が関連付けられている管理オブジェクトコンテキストを知っています。逆に管理オブジェクトコンテキストも、どの管理オブジェクトが自身に属するかを知っています。しかしデフォルトでは、管理オブジェクトとそのコンテキストとの参照関係は弱く、管理メモリ環境ではどちらのオブジェクトも、もう一方のオブジェクトを、「関係」先として保持していません。

したがって一般に、コンテキストがあっても、管理オブジェクトインスタンスの寿命が尽きていない、と保証することはできません。同様に、管理オブジェクトが存在していても、コンテキストが生存しているとは限りません。すなわち、オブジェクトをフェッチできても、将来にわたって存在するとは限らないのです。参照カウント方式のアプリケーションでは、特別な処理をしていなければ、管理オブジェクトの寿命がイベントループで決まります。自動解放された管理オブジェクトは、イベントループの自動解放プールを解放する時点で割り当て解除されます。

この規則には例外があります。管理オブジェクトコンテキストは、変更（挿入、削除、更新）されたオブジェクトへの強い参照を、保留になっているトランザクションがコミット（save:）または破棄（reset/rollback）されるまでの間、管理（参照カウント環境では保持）しています。さらに、取り消しマネージャも、変更されたオブジェクトを保持しています（「変更/取り消し管理」（77 ページ）を参照）。

コンテキストのデフォルトの振る舞いを変更して、管理オブジェクトを保持するようにすることができます。それには、引数をYESとしてsetRetainsRegisteredObjects:メッセージを送ります。これにより管理オブジェクトの寿命は、コンテキストの寿命に依存して決まるようになります。これは、メモリ中にキャッシュしているデータセットが小さい場合に便利でしょう。たとえば、シート内で編集しているときのように、コンテキストが、単一のイベントサイクルを越えて存続しうるオブジェクトの一時的なセットを制御している場合です。また、複数のスレッドがあって、相互にデータをやり取りしている場合にも有用です。たとえば、バックグラウンドでフェッチ操作を行い、オブジェクトIDを主スレッドに渡す、というような状況です。バックグラウンドのスレッドは、事前にフェッチしたオブジェクトを主スレッドに渡せるよう、保持しておかなければなりません。主スレッドが実際にオブジェクトIDを使って、ローカルインスタンスをフォールト状態にしたことが分かるまでの間、これが必要です。

通常、実際に必要な管理オブジェクトのみを保持するためには、独立したコンテナを使う必要があります。配列や辞書のほか、オブジェクトコントローラ（たとえばNSArrayControllerのインスタンス）を使って、自身が管理するオブジェクトを明示的に保持できます。必要のない管理オブジェクトは、関係が解消されるなど、割り当て解除可能な状態になった時点で、実際に解除されます。

管理オブジェクトコンテキストが不要になった、あるいは何らかの理由でコンテキストをその永続ストアコーディネータから「切り離し」たくなった場合でも、コンテキストのコーディネータをnilにする、という方法は*使わないで*ください。

```
// 例外が発生する
[myManagedObjectContext setPersistentStoreCoordinator:nil];
```

代わりに、単にコンテキストの所有権を放棄し（管理メモリ環境ではreleaseメッセージを送信）、正常に割り当て解除されるようにします。

関係の循環参照の切断

管理オブジェクトどうしが関係で結ばれている場合、各オブジェクトは、関係先オブジェクトの強い参照を保持しています。管理メモリ環境では、これにより循環参照が生じ（「オブジェクトの所有権と破棄」を参照）、不要なオブジェクトを割り当て解除できなくなることがあります。循環参照を確実に切断するため、オブジェクトが不要になったときに、管理オブジェクトコンテキストのrefreshObject:mergeChanges:メソッドでフォールト状態にする、という方法があります。

通常はrefreshObject:mergeChanges:で、管理オブジェクトのプロパティ値をリフレッシュします。mergeChangesフラグがYESであれば、このメソッドは、このオブジェクトのプロパティ値に、永続ストアコーディネータにあるオブジェクトのプロパティ値をマージします。一方、NOであれば、マージせず単にフォールト状態に戻すようになっており、したがって関係する管理オブジェクトは解放されます。これにより、管理オブジェクトとそれが保持する管理オブジェクトとの間の循環参照を切断できるのです。

なお、当然ながら、オブジェクトが割り当て解除されるためには、グラフ外にも当該オブジェクトを保持しているものがあってはなりません。「変更/取り消し管理」（77 ページ）も参照してください。

変更/取り消し管理

変更（挿入、削除、更新）を保留している管理オブジェクトは、当該コンテキストに`save:`、`reset`、`rollback`、`dealloc`のいずれかのメッセージが送られるか、必要な回数の「取り消し」操作により変更が取り消されるまで、そのコンテキストに保持されています。

*コンテキストに関連付けられた取り消しマネージャは、変更を施された管理オブジェクトを記憶しています。*デフォルトでは、コンテキストの取り消しマネージャは、無限に取り消し/再実行スタックを維持しておけるようになっています。アプリケーションのメモリ消費量を抑えるため、適宜、コンテキストの取り消しスタックをクリアしてください（`removeAllActions`を使用）。コンテキストの取り消しマネージャを保持していなければ、コンテキストから割り当て解除されます。

Core Dataの取り消し管理機能を使わない場合は、コンテキストの取り消しマネージャとして`nil`を設定することにより、アプリケーションの動作に必要な資源を減らすことができます。これは特に、バックグラウンドのワーカスレッドや、大容量のインポート操作、バッチ処理の場合に有用です。

関係とフェッチ済みプロパティ

関係を定義する際には、判断するべき事項がいくつもあります。関係で結ばれる先のエンティティは何か、対1か対多か、必須か否か、といった事項に加え、対多の場合は関係先オブジェクトの最大/最小数を設定するか、関係「元」オブジェクトが削除されたときどうするか、などを決めなければなりません。決めた事項はすべて、モデルの形で表現します。特に興味深いと思われるのは、「多対多」関係の表し方でしょう。モデル化の方法は2通りあり、スキーマの意味づけに応じて選ぶことになります。

オブジェクトグラフに修正を施す際は、参照の整合性を保つことが重要です。Core Dataでは、整合性を損なうことなく、容易に管理オブジェクト間の関係を変更できるようになっています。この辺りの動作の多くは、管理オブジェクトモデルに指定された、「関係記述」というものによって決まります。

Core Dataでは、ストアをまたがる関係は生成できません。あるストア内にあるオブジェクトから、別のストア内のオブジェクトに向かう関係を表現したい場合は、「フェッチ済みプロパティ」を使います。

モデルにおける関係の定義

管理オブジェクトモデルでは、関係の定義自体は単純ですが、いくつかの事項を適切に指定する必要があります。重要な指定事項としては、関係の名前、関係で結ばれる先のエンティティ、**基数**（対1関係か対多関係か、の区別）があります。しかし、オブジェクトグラフの整合性という観点からは、「逆関係」と「削除規則」が重要です。その関係が必須か否か、および（対多関係の場合）最大数/最小数も、グラフの妥当性に影響を及ぼします。

関係についての基本事項

「関係」は、関係で結ばれる先のオブジェクトが属する、エンティティまたは親エンティティを指定する、という形で定義します。関係「先」エンティティが、関係「元」エンティティと同じであっても構いません（**反射的**関係）。また、関係は「均質」でなくても構いません。たとえば、Employeeエンティティに、Manager、Flunkyという2つのサブエンティティがある場合、ある部署の関係「先」となる従業員は、Employee（抽象エンティティではないと想定）、Manager、Flunkieのいずれであってもよいのです。

さらに、関係が対1、対多のどちらであるかを指定できます。対1関係は、関係先オブジェクトの参照で表します。対多関係はミュータブルな集合として表します（ただしフェッチ済みプロパティの場合は配列）。対1および対多は、普通には「1対1」、「1対多」と呼ばれる関係を暗に表しています。「多対多」の関係は、関係とその逆がどちらも対多であるものです。これについては考慮するべき事項がいくつかあるので、「多対多の関係」（82 ページ）で詳しく説明します。

また、対多関係の関係先オブジェクトの個数には、上限と下限を設定できます。下限を0にすることはできません。たとえば、各部署の従業員数は3人以上40人以下、というような指定ができます。これとは別に、関係には、必須か否かの指定も可能です。必須であれば、関係先として少なくとも1つオブジェクトを指定しなければ、妥当ではなくなります。

「基数」と「必須か否か」とは直交する概念です。上限や下限を指定したとしても、これとは別に、「必須ではない」旨を指定できます。すなわち、関係先オブジェクトがなくても構わないけれども、ある場合はその個数が所定の範囲内でなければならない、ということです。

重要なのは、関係を定義したからと言って、関係元のオブジェクトを生成したとき、関係先オブジェクトも生成されるわけではない、ということです。この点では、関係の定義は、標準的なObjective-Cのクラスにおけるインスタンス変数の宣言に似ています。次の例を考えてみましょう。

```
@interface Widget : NSObject
{
    Sprocket *sprocket;
}
```

Widgetのインスタンスを生成しても、（たとえば`init`メソッドをオーバーロードして）適当な処理を記述していない限り、Sprocketのインスタンスが生成されるわけではありません。同じように、Addressエンティティと、EmployeeからAddressへの関係（必須、対1）を定義しても、Employeeのインスタンスを生成しただけで、Addressのインスタンスが生成されることはないのです。必須の対多関係を定義し、その最小数を`1`とした場合も同様です。

逆関係

関係の多くは、元来、両方向です。あるDepartment（部署）に、当該部署に属するEmployee（従業員）との対多関係があれば、EmployeeからDepartmentに向かう「逆の」関係もあるのです。その重要な例外として、弱い片方向の関係を表す、フェッチ済みプロパティがあります。関係先から関係元に向かう「逆」関係はありません（「フェッチ済みプロパティ」（86 ページ）を参照）。

通常は関係を両方向としてモデル化し、逆関係を適切に指定してください。Core Dataはこの情報を使って、オブジェクトグラフに変更が施されたとき、整合性を確認するようになっています（「関係の操作とオブジェクトグラフの整合性」（81 ページ）を参照）。両方向の関係としてモデル化するのが適切でない状況と、その場合に生じうる問題点については、「片方向の関係」（85 ページ）を参照してください。

関係の削除規則

関係の削除規則は、関係元オブジェクトを削除しようとしたときに何が起こるか、を指定するものです。「削除したとき」ではなく「削除しようとしたとき」である点に注意してください。削除規則がDenyであれば、関係元オブジェクトが削除されない可能性もあります。ここでも部署と従業員の関係を例として、削除規則ごとの動作を説明します。

Deny

関係で結ばれる先に1つでもオブジェクトがあれば、関係元オブジェクトは削除されません。

たとえば、ある部署を削除するためには、当該部署に属する従業員を全員異動させる（または解雇する）必要があります。

Nullify
: 関係先から関係元に向かう逆関係をnullにします。

  たとえばある部署を削除すると、当該部署に属していた従業員は全員、所属部署がなくなります。この処理で問題が起こらないのは、従業員から部署への関係が必須ではないか、次に保存するまでの間に、全員を新しい部署に所属させた場合に限ります。

Cascade
: 関係先のオブジェクトを（連鎖的に）削除します。

  たとえばある部署を削除すると、当該部署に属していた従業員は全員解雇されます。

No Action
: 関係先オブジェクトに対しては何もしません。

  たとえばある部署を削除しても、当該部署に属していた従業員はそのままです。元の部署にまだ所属しているように見えてしまうかもしれませんが、それでも構わない、という規則です。

明らかに「No Action」以外の規則は、役に立つ状況が異なります。どの規則が適切かは、ビジネスロジックに応じて、開発者が判断しなければなりません。一方、「No Action」規則が役に立つ状況は、それほど明らかではないでしょう。オブジェクトグラフの整合性が損なわれる（たとえば、従業員から、存在しない部署への関係が残ってしまう）恐れがあるからです。

「No Action」規則を指定した場合、オブジェクトグラフの整合性を維持するのは開発者の責任です。逆関係として適切な値を設定することも含みます。対多関係で結ばれた先のオブジェクト数が膨大になる場合は、この規則が役に立つかもしれません。

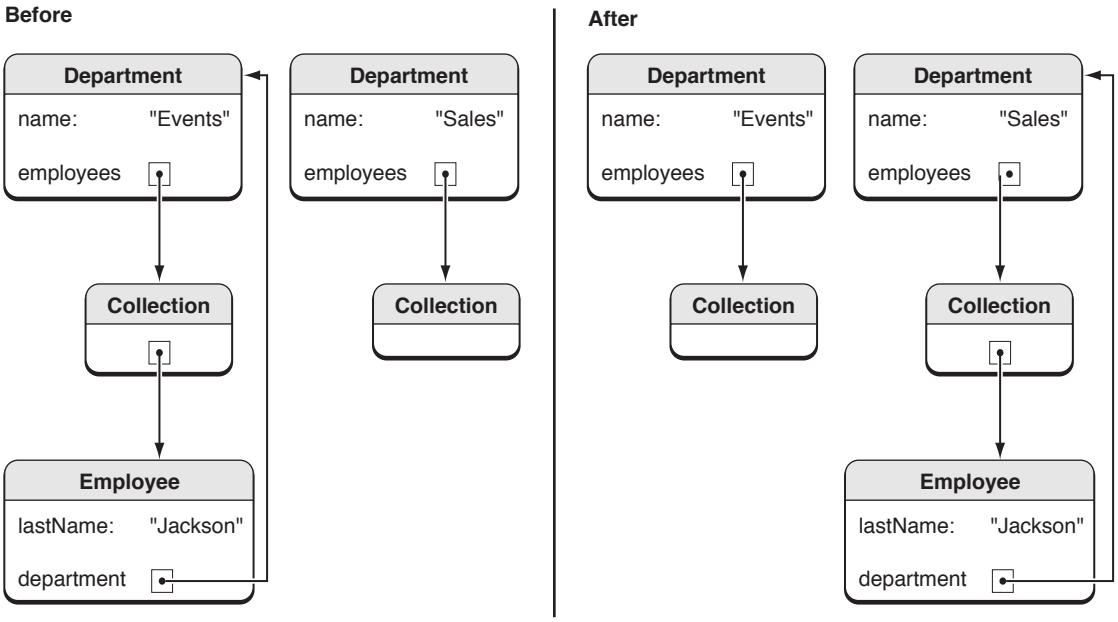
関係の操作とオブジェクトグラフの整合性

一般に、プログラムで関係を操作すること自体は簡単です。その例を「プロパティのアクセスと修正」（65 ページ）に示しました。

Core Dataにはオブジェクトグラフの整合性を維持するための機能が組み込まれているので、関係で結ばれた一方のオブジェクトを変更するだけで済んでしまいます。これは、対1および対多のほか、「多対多」の関係にも成り立ちます。次の例を考えてみましょう。

従業員から上長への関係があれば、当然に、当該上長からその従業員への逆関係があります。新しい従業員をある上長の下に配備すれば、上長はこの従業員に関して管理責任を負うことになるからです。したがって、この新しい従業員を、部下のリストに追加しなければなりません。同様に、従業員がある部署から別の部署に異動した場合、図 1（82 ページ）に示すように、あちこちに修正を施す必要があります。すなわち、従業員が属する新しい部署を設定し、元の部署の従業員リストから削除し、新しい部署の従業員リストに追加しなければなりません。

図 1　従業員を新しい部署に異動



Core Dataフレームワークがなかったとすれば、オブジェクトグラフの整合性を維持するために、相当量のコードを記述しなければならなかったでしょう。さらに、Departmentクラスの実装を調べて、逆関係をどのように設定するか判断しなければなりません（開発が進むにつれて判断が変わる可能性もあります）。Core Dataフレームワークを導入していれば、次の1行だけで済んでしまいます。

```
anEmployee.department = newDepartment;
```

あるいは次のように記述しても構いません。

```
[newDepartment addEmployeeObject:anEmployee];
```

このような書き方ができる理由については、「管理オブジェクトのアクセサメソッド」（45 ページ）を参照してください。どちらも実質的に同じ効果があります。フレームワークは管理オブジェクトモデルを参照して、オブジェクトグラフの現在の状態から、どの関係を確立しどの関係を切断するか、自動的に判断します。

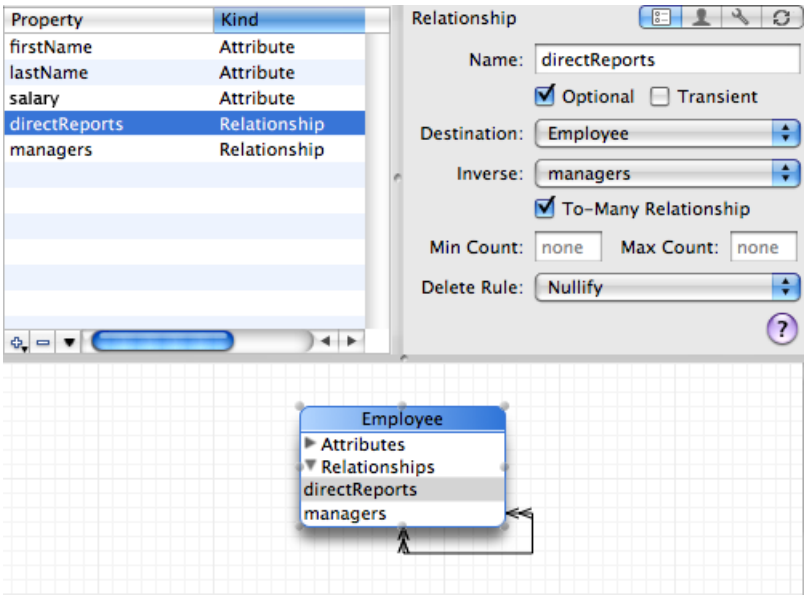
多対多の関係

「多対多」の関係は、対多関係を2つ組み合わせて定義します。一方の対多関係は、第1のエンティティから第2のエンティティに向かうものです。もう一方は、第2のエンティティから第1のエンティティに向かいます。互いに逆関係になるよう設定してください（データベース管理の経験があれば不安になるかもしれませんが、心配は要りません。SQLiteストアであれば、中間ジョインテーブルが自動的に作成されます）。

重要： 「多対多」の関係は両方向に定義しなければなりません。すなわち、2つの関係を指定し、互いに逆関係になるようにします。片方向の対多関係だけ定義して、「多対多」の関係として使うことはできません。参照整合性の面で問題が生じることになります。

この方法は、同じエンティティとの関係（「反射的」関係）についても使えます。たとえば、1人の従業員に複数の上長がいてもよい（そしてもちろん、上長から見れば直属の部下が複数いる）場合、Employeeからそれ自身に向かう対多関係「`directReports`」を定義します。これは、やはりEmployeeからそれ自身に向かう、もうひとつの対多関係「`employees`」の逆関係です。この様子を図 2 （83 ページ）に示します。

図 2 反射的な多対多の関係の例

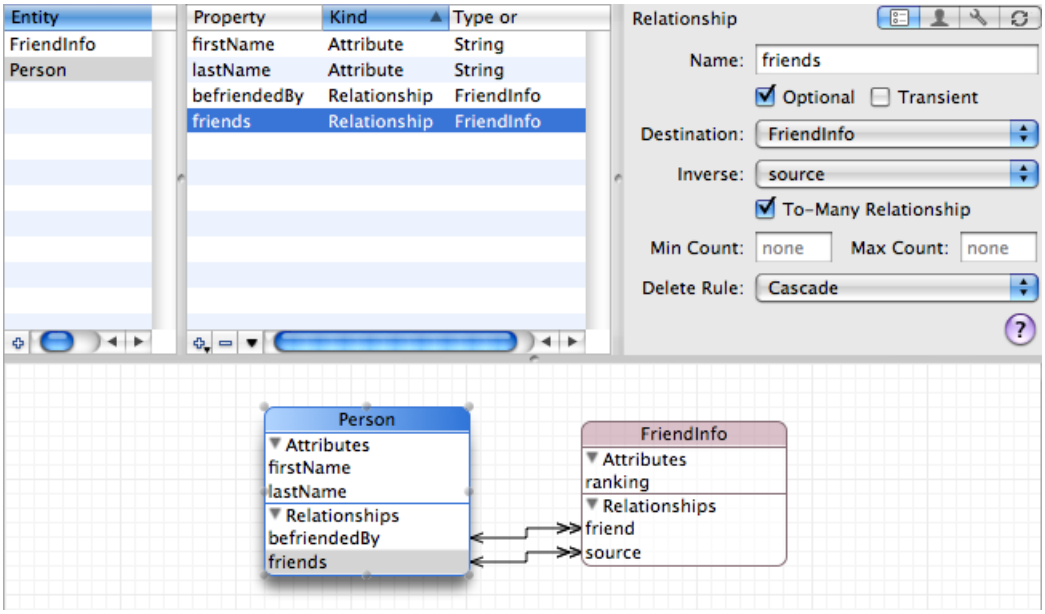


関係をそれ自身の逆関係と見なすことも可能です。たとえばPersonエンティティに「`cousins`」（従兄弟）関係を定義すれば、これは自分自身の逆関係でもあります。

重要： Mac OS X v10.4では、（「cousins」のように）関係が自分自身の逆関係でもあるような「多対多」関係を、SQLiteストアでうまく扱えません。

関係については、その意味や、どのようにモデル化するか、ということも考えなければなりません。「多対多」であって、自分自身の逆でもあるような関係をモデル化した例として、「friends」（友人）関係を考えてみましょう。ある人物は、（好むと好まざるとにかかわらず）その従兄弟から見ればやはり従兄弟であるのに対し、その友人から見て、やはり友人であるとは限りません。このような関係については、中間（「ジョイン」）エンティティを使います。中間エンティティには、その関係に別の情報を追加するためにも使える、という利点があります。たとえば「FriendInfo」エンティティには、親しさの度合いを表す「ranking」属性を定義できます。その様子を図3 （84 ページ）に示します。

図 3　中間エンティティを使って「friends」関係を表すモデル



この例で、PersonにはFriendInfoとの対多関係が2つあります。「friends」はある人物の友人、「befriendedBy」はその人物を友人と思っている人を表します。FriendInfoは、「ある方向の」友人関係に関する情報を表します。そのインスタンスには、友人関係「元」に当たる人物（source）と、その人物が友人と思っている人物（friend）が記述されています。互いに友人であると思っている場合は、「source」と「friend」が入れ替わったインスタンスが別にあるはずです。このようなモデルについては、ほかにも考慮するべき事項があります。

- ある人物から別の人物への「友人」関係を設定するためには、FriendInfoのインスタンスを生成する必要があります。2人がお互いに友人と思っている場合は、FriendInfoのインスタンスを2つ生成しなければなりません。
- 友人関係を解消するためには、FriendInfoの該当するインスタンスを削除する必要があります。
- PersonからFriendInfoへの関係の削除規則はCascadeとなります。ある人物をストアから削除すれば、FriendInfoのインスタンスも無意味になるので、削除しなければなりません。

  このことからすぐに分かるように、FriendInfoからPersonへの関係は、必須と設定しなければなりません。FriendInfoのインスタンスは、「source」または「friend」がnullであれば意味のないものになってしまいます。
- ある人物の友人を調べたい場合は、次のように、「friends」関係で結ばれたFriendInfoの、「friend」の先にある人物をすべて集約することになります。

  ```
  NSSet *personsFriends = [aPerson valueForKeyPath:@"friends.friend"];
  ```

  一方、ある人物を友人と思っている人を調べたい場合は、次の例のように、「befriendedBy」関係で結ばれたFriendInfoの、「source」の先にある人物を集約します。

  ```
  NSSet *befriendedByPerson = [aPerson valueForKeyPath:@"befriendedBy.source"];
  ```

片方向の関係

関係は、どうしても両方向でモデル化しなければならないわけではありません。そうしない方がよい場合もあります。たとえば、対多関係で結ばれるオブジェクト数が非常に多く、しかも関係を横断する処理が滅多にないような状況では、多数の関係先オブジェクト（フォールト状態）が無駄に実体化することのないよう、片方向でモデル化した方がよいかもしれません。ただしその場合、オブジェクトグラフの整合性を維持し、変更を追跡し、取り消し処理に備えるのは、開発者の責任になります。したがってこの方法はあまりお勧めできません。片方向のモデル化が有用なのは、一般に、対1関係の場合だけです。

逆関係の指定がない、片方向の関係を使ったモデルでは、オブジェクトグラフの整合性が崩れる恐れがあります。

片方向の関係でモデル化したために問題が起こる例を示します。2つのエンティティEmployeeおよびDepartmentと、EmployeeからDepartmentに向かう対1関係「department」があるとします。この関係は必須で、削除規則は「Deny」です。逆関係はありません。ここで、次のコードを考えてみましょう。

```
Employee *employee;
Department *department;
// エンティティのインスタンスは正常に生成されたと仮定
[employee setDepartment:department];
[managedObjectContext deleteObject:department];
BOOL saved = [managedObjectContext save:&error];
```

（関係は必須であるにもかかわらず）employeeが別途変更されない限り、保存に成功してしまいます。Employee.department関係に逆がないため、departmentを削除しても、employeeに変更が施された旨の印がつきません（したがって、保存の際に検証が行われません）。

ここで、次のコードを追加したとします。

```
id x = [employee department];
```

xは、nilではなく、「どこでもない」部署（意味のない値）になってしまいます。

これに対し、「department」関係に逆があれば（そして削除規則が「No Action」でなければ）、削除した旨が伝搬し、employeeに値が変化した旨の印がつくので、想定通りに処理が進みます。

このような事情があるので、一般に、片方向の関係を使うのは避けてください。両方向の関係であれば、オブジェクトグラフを適切に維持するために必要な情報が、フレームワークに与えられます。それでも片方向の関係を使いたいのであれば、整合性を維持するためのコードを記述する必要があります。上記の例では、

```
[managedObjectContext deleteObject:department];
```

というコードの後に、次の記述が必要です。

```
[employee setValue:nil forKey:@"department"]
```

このようになっていれば、関係は必須である、という設定に反するので、保存操作は想定通り失敗します。

ストアをまたがる関係

ある永続ストアのインスタンスから、別の永続ストアにあるインスタンスへの関係は、定義しないようにしてください。Core Dataは、この機能には未対応です。異なるストアのエンティティ間に関係を定義したい場合は、フェッチ済みプロパティを使う必要があります（「フェッチ済みプロパティ」（86 ページ）を参照）。

フェッチ済みプロパティ

フェッチ済みプロパティは、弱い片方向の関係を表します。従業員と部署の例では、部署の「フェッチ済みプロパティ」として、「最近雇用した従業員」が考えられます（従業員の側から出る、逆向きの関係はありません）。一般に、フェッチ済みプロパティは、ストアをまたがる関係、「緩やかに結合した」関係、一時的なグループ化などのモデル化に向いています。

フェッチ済みプロパティは関係に似ていますが、いくつか重要な違いがあります。

- フェッチ済みプロパティは、「直接」の関係ではなく、フェッチ要求により値を求めるようになっています（フェッチ要求では通常、結果を制限するために「述語」を使います）。
- フェッチ済みプロパティは、集合ではなく配列で表現されます。プロパティの値を取得するフェッチ要求には整列順を指定できるので、フェッチ済みプロパティも順序づけが可能です。
- フェッチ済みプロパティは遅延評価され、その後キャッシュされます。

フェッチ済みプロパティは、ある面でスマート再生リストにも似ていますが、動的に求められるのではない、という重要な制約があります。結び付けられた先のエンティティのオブジェクトが変化した場合、フェッチ済みプロパティを再評価して、最新の状態にしなければなりません。`refreshObject:mergeChanges:`を使って、強制的にプロパティをリフレッシュします。その結果、フォールト状態のオブジェクトが次に発動した時点で、プロパティに対応するフェッチ要求が再実行されます。

フェッチ済みプロパティの述語に記述できる特殊変数として、`$FETCH_SOURCE`と`$FETCHED_PROPERTY`の2つがあります。`$FETCH_SOURCE`はこのプロパティを持つ管理オブジェクトを表します。これをキーパスで修飾し、たとえば「`university.name LIKE [c] $FETCH_SOURCE.searchTerm`」のように記述できます。`$FETCHED_PROPERTY`はエンティティのフェッチ済みプロパティ記述を表します。プロパティ記述にはユーザ情報辞書が付随しており、キーと値の組の形で、どのような情報でも格納しておけます。フェッチ済みプロパティの述語は、この変数を使って記述しておけば、実際に適用される式を動的に変えることができます。当該オブジェクトと関係で結ばれたオブジェクトも、（キーパスを介して）変更できます。

変数置換の様子を見るため、あるフェッチ済みプロパティを考えてみましょう。結び付けられた先のエンティティはAuthorで、「`(university.name LIKE [c] $FETCH_SOURCE.searchTerm) AND (favoriteColor LIKE [c] $FETCHED_PROPERTY.userInfo.color)`」という述語が設定されているとします。このプロパティを持つオブジェクトの属性`searchTerm`が「Cambridge」で、フェッチ済みプロパティのユーザ情報辞書には、「color」をキーとして「Green」という値が設定されているとします。すると述語は「`(university.name LIKE [c] "Cambridge") AND (favoriteColor LIKE [c] "Green")`」となります。これは、「Cambridge」大学に所属し好みの色が「Green」であ

るAuthorに合致します。仮に `searchTerm`属性の値を「Durham」と変更したとすれば、述語は「`(university.name LIKE [c] "Durham") AND (favoriteColor LIKE [c] "Green")`」に変わります。

もっとも、置換によって述語の構造を変えることはできない、という重要な制約があります。たとえば、LIKE述語を複合述語に変更することも、演算子（この例では「`LIKE [c]`」）を変更することもできません。さらに、Mac OS X v10.4では、XMLストアおよびBinaryストアでしか動作しません。SQLiteストアの場合、適切なSQLが生成できないからです。

非標準の永続属性型

Core Dataでは、永続属性の値として、文字列、日付、数値など標準的な型が使えるようになっています。しかし、直接の支援機能がない型の属性値を扱いたいことも考えられます。たとえばグラフィックスアプリケーションでは、Rectangleエンティティの`color`属性、`bounds`属性として、`NSColor`や`NSRect`という構造体のインスタンスを使えれば便利かもしれません。この章では、非標準の属性型を扱う方法として、変換可能な属性を用いる方法と、一時プロパティを用いる方法の2通りを説明します。いずれも、内部では標準型の永続プロパティとして管理し、外部に対しては、望ましい（非標準）型に見えるよう細工します。

はじめに

永続属性は、Core Dataフレームワークが認識できる型でなければ、永続ストアに適切に格納し、検索することはできません。Core Dataには、文字列、日付、数値など、永続属性の値によく使われる型の支援機能が組み込まれています（`NSAttributeDescription`を参照）。しかし、直接の支援機能がない、色、Cの構造体などといった型を使いたい場合もあるでしょう。

永続属性に非標準の型を使うためには、変換可能な属性または*一時*プロパティを用います。どちらの方法も考え方は同じで、エンティティの「消費者」に対しては（非標準の）望ましい型の属性のように見せ、「背後では」Core Dataが管理できる型に変換するのです。しかし、変換可能な属性の場合、該当する属性そのものだけを定義し、変換は自動処理に任せます。一方、一時プロパティの場合、該当する属性と「一時」属性の2つを用意し、相互に変換するコードも記述しなければなりません。

変換可能な属性

「変換可能な属性」の考え方は、非標準の型として属性にアクセスできるようにしつつ、Core Dataはその背後で、標準の型との間で相互に変換を行う、というものです。変換処理には`NSValueTransformer`のインスタンスを使い、また、変換元/先の標準型は`NSData`のインスタンスとします。永続ストアには、変換されたデータインスタンスを格納します。

デフォルトでは、`NSKeyedUnarchiveFromDataTransformerName`で表される変換器を使うようになっていますが、必要ならば独自の変換器を指定することも可能です。カスタム変換器を指定する場合は、非標準型のインスタンスと`NSData`のインスタンスとの間で、相互に変換できるものでなければなりません。デフォルトの変換器を用いる場合は、変換器の名前を指定しないでください。

> **重要：** デフォルトの変換器とは、`NSKeyedUnarchiveFromDataTransformerName`で表される変換器のことですが、実際には逆方向の変換に使われます。デフォルトの変換器を明示的に指定すると、間違った方向の変換に使われることになってしまいます。

属性が変換可能である旨と、実際に用いる変換器の名前を、Xcodeのモデルエディタで、またはプログラム上で指定します。

- Xcodeのモデルエディタを使う場合は、属性の「Type」ポップアップメニューから「Transformable」を選択し、「Value Transformer Name」欄に名前を入力してください。
- プログラム上で指定する場合は、`NSTransformableAttributeType`を引数として`setAttributeType:`を実行し、次に（独自の変換器を指定する場合は）`setValueTransformerName:`で変換器の名前を設定します。

ほかには何もしなくても構いません。しかし実際には、コンパイラの警告を抑制するため、次のように属性のプロパティを宣言するとよいでしょう（`favoriteColor`に着目）。

```
@interface Person : NSManagedObject
{
}

@property (nonatomic, retain) NSString * firstName;
@property (nonatomic, retain) NSString * lastName;

@property (nonatomic, retain) NSColor * favoriteColor;

@end
```

コンパイラの警告を抑制するため、実装ディレクティブも追加してください。

```
@implementation Person

@dynamic firstName;
@dynamic lastName;

@dynamic favoriteColor;

@end
```

するとこの属性は、次のコード例のように、標準型の属性と同様に使えるようになります。

```
Employee *newEmployee =
    [NSEntityDescription insertNewObjectForEntityForName:@"Employee"
        inManagedObjectContext:myManagedObjectContext];

newEmployee.firstName = @"Captain";
newEmployee.lastName = @"Scarlet";
newEmployee.favoriteColor = [NSColor redColor];
```

カスタムコード

以下の節では、オブジェクト値とスカラ値を属性型として使う実装例を示します。いずれの場合も、まず永続属性を指定しなければなりません。

注： オブジェクト値の項では、例として`NSColor`のインスタンスを使っていますが、Mac OS X v10.5の場合、これは変換可能な属性として実装してください。

基本的なアプローチ

非標準型の属性を扱えるようにするため、管理オブジェクトモデルで2つの属性を定義します。ひとつは実際に使いたい型の属性です（たとえばcolorオブジェクト、rectangle構造体を値とする型）。これは一時属性です。もうひとつは、当該属性の「影の」表現です。これは永続属性です。

一時属性の型は、未定義（`NSUndefinedAttributeType`）としておきます。この一時属性を格納、検索する必要はないので、実装においては、どのようなオブジェクト型の値でも使えます。一方、Core Dataは一時属性の状態も追跡するようになっているので、オブジェクトグラフの管理機能（取り消し/再実行の支援機能など）はこの属性に対しても有効です。

影の属性の型は、標準の「具象」型の中から選ばなければなりません。次に、カスタム管理オブジェクトクラスを定義し、一時属性のアクセサメソッドを実装します。このメソッドは、対応する永続属性から値を取得して返す、あるいは永続属性に値を格納する、という動作をします。

基本的なアプローチは、オブジェクト値でもスカラ値でも同じです。非標準のデータ型を、いずれかの標準データ型で表現する方法を考えなければなりません。しかし、スカラ値の場合は、ほかにも制約があります。

スカラ値の場合の制約

実装するアクセサメソッドは、キー値コーディング（およびキー値監視）に準拠する必要があります。キー値コーディングは、`NSPoint`、`NSSize`、`NSRect`、`NSRange`という、ごく限られた構造体にしか対応していません。

Core Dataで直接扱えず、キー値コーディングも対応していないスカラ値や構造体を使いたい場合、管理オブジェクトにオブジェクトとして格納する必要があります。このオブジェクトは、通常は`NSValue`のインスタンスですが、カスタムクラスを定義しても構いません。すると、後述のようなオブジェクト値として扱えるようになります。検索の際、`NSValue`（またはカスタム）オブジェクトからスカラ値や構造体を取り出す処理や、設定の際、スカラ値や構造体を`NSValue`（またはカスタム）オブジェクトに変換する処理は、オブジェクトを使う側で実装しなければなりません。

永続属性

非標準の属性型を扱う場合、（影で）値を格納するために使う、標準の属性型を選択しなければなりません。対象とする（非標準の）属性型と、相互に変換する手段に応じて、適当な型を選んでください。多くの場合、アーカイバを利用すれば、`NSData`オブジェクトに変換するのは容易です。たとえば下記のコード例のようにすれば、「色」オブジェクトをアーカイブできます。属性を`NSValue`やカスタムクラスのインスタンスとして表している場合も、同じ方法が使えます（もちろんカスタムクラスの場合は、`NSCoding`プロトコルに準拠しているか、標準のデータ型に変換する手段を提供しなければなりません）。

```
NSData *colorAsData = [NSKeyedArchiver archivedDataWithRootObject:aColor];
```

変換にはどのような手段を使っても構いません。たとえばNSRect構造体を、文字列オブジェクトに変換する、という方法も考えられます（文字列はもちろん永続ストアに格納可能）。

```
NSRect aRect; // インスタンス変数
NSString *rectAsString = NSStringFromRect(aRect);
```

逆に文字列からrectangleに変換する際には、NSRectFromStringを使います。変換処理は頻繁に起こるので、できるだけ効率的な方法を検討してください。

通常、永続属性にアクセスするためのカスタムメソッドは必要ありません。これは実装詳細であり、エンティティ自身以外がアクセスすることはないからです。この値を直接修正すると、エンティティオブジェクトの整合性が損なわれる恐れがあります。

オブジェクト属性

非標準の属性がオブジェクト値の場合、管理オブジェクトモデルでは、型が未定義の一時属性である旨を指定します。エンティティのカスタムクラスを実装する際、属性に対応するインスタンス変数を追加する必要はありません。管理オブジェクト内部のプライベートなストアが使えます。以下に示す実装例では、一時値をキャッシュしている点に注意してください。これにより、効率的に値にアクセスできるようになります。また、変更管理のためにも必要です。カスタムインスタンス変数を定義する場合、そのクリーンアップ処理は、deallocやfinalizeではなく、didTurnIntoFaultで行う必要があります。

一時値の取得や設定には、2通りの戦略があります。一時値の取得については、「遅延」取得（あるいはオンデマンドの取得、「遅延型の取得アクセサ」（92ページ）を参照）と、awakeFromFetch中に取得（「事前計算型の取得アクセサ」（93ページ）を参照）する方法の、2通りがあります。値が大容量の場合（ビットマップなど）は、遅延取得の方が優れています。永続値の更新については、一時値が変化するたびに更新する方法（「即時更新型の設定アクセサ」（93ページ）を参照）と、オブジェクトを保存する時点まで更新を遅らせる方法（「遅延更新型の設定アクセサ」（94ページ）を参照）があります。

遅延型の取得アクセサ

取得アクセサでは属性値を、管理オブジェクト内部のプライベートなストアから検索します。値がnilであれば、まだキャッシュしていない可能性があるので、対応する永続値を検索します。それがnilでなければ、適当な型に変換し、さらにこの値をキャッシュします。（設定メソッドに対応したキー値監視変更通知メソッドを起動する必要はありません。これは値の変更を表すのではないからです）。「色」属性を取得する、遅延型のアクセサの例を以下に示します。

```
- (NSColor *)color
{
    [self willAccessValueForKey:@"color"];
    NSColor *color = [self primitiveColor];
    [self didAccessValueForKey:@"color"];
    if (color == nil)
    {
        NSData *colorData = [self colorData];
        if (colorData != nil)
        {
            color = [NSKeyedUnarchiver unarchiveObjectWithData:colorData];
            [self setPrimitiveColor:color];
        }
    }
```

```
    return color;
}
```

事前計算型の取得アクセサ

このアプローチでは、awakeFromFetch内であらかじめ永続値を検索し、キャッシュしておきます。（設定メソッドに対応したキー値監視変更通知メソッドを起動する必要はありません。これは値の変更を表すのではないからです）。

```
- (void)awakeFromFetch
{
    [super awakeFromFetch];
    NSData *colorData = [self colorData];
    if (colorData != nil)
    {
        NSColor *color;
        color = [NSKeyedUnarchiver unarchiveObjectWithData:colorData];
        [self setPrimitiveColor:color];
    }
}
```

取得アクセサでは、単にこのキャッシュされた値を返します。

```
- (NSColor *)color
{
    [self willAccessValueForKey:@"color"];
    NSColor *color = [self primitiveColor];
    [self didAccessValueForKey:@"color"];
    return color;
}
```

属性に頻繁にアクセスする場合はこの方法が有利です。取得アクセサには条件分岐がありません。

即時更新型の設定アクセサ

この設定アクセサでは、一時属性と永続属性の両方に、同時に値を設定します。非標準型の値は、標準型に変換した上で、永続値として設定します。このとき、キー値監視の変更通知メソッドを起動して、管理オブジェクトを監視しているオブジェクト（管理オブジェクトコンテキストを含む）に、修正した旨を通知しなければなりません。「色」属性の設定アクセサの例を以下に示します。

```
- (void)setColor:(NSColor *)aColor
{
    [self willChangeValueForKey:@"color"];
    [self setPrimitiveValue:aColor forKey:@"color"];
    [self didChangeValueForKey:@"color"];
    [self setValue:[NSKeyedArchiver archivedDataWithRootObject:aColor]
                forKey:@"colorData"];
}
```

この方法は、一時値を更新するたびに、永続値を再計算する必要があるため、性能の面では劣ります。

遅延更新型の設定アクセサ

この設定アクセサは、一時属性の値しか設定しません。代わりに、オブジェクトを保存する直前にのみ永続値を更新するため、`willSave`メソッドを実装します。（設定メソッドに対応したキー値監視変更通知メソッドを起動する必要はありません。これは値の変更を表すのではないからです）。

```
- (void)setColor:(NSColor *)aColor
{
    [self willChangeValueForKey:@"color"];
    [self setPrimitiveValue:aColor forKey:@"color"];
    [self didChangeValueForKey:@"color"];
}

- (void)willSave
{
    NSColor *color = [self primitiveValueForKey:@"color"];
    if (color != nil)
    {
       [self setPrimitiveValue:[NSKeyedArchiver archivedDataWithRootObject:color]
                forKey:@"colorData"];
    }
    else
    {
        [self setPrimitiveValue:nil forKey:@"colorData"];
    }
    [super willSave];
}
```

この方法では、必須か否かを指定する際に注意が必要です。色が必須属性である場合、（別途何らかの手段を講じていなければ）一時属性は必須にしますが、永続属性の方は、必須としてはなりません。こうしないと、1回目の保存の際、検証エラーが発生する恐れがあります。

オブジェクトを生成した時点では、永続属性`colorData`の値は`nil`です。色を更新しても、`colorData`属性は`nil`のままです。保存の際には、`willSave`よりも前に`validateForUpdate:`が起動されます。この時点では、`colorData`の値は`nil`のままなので、検証エラーになってしまうのです。

スカラ値

プロパティをスカラ値として宣言することは可能ですが、Core Dataはそのアクセサメソッドを動的に生成できないので、独自に実装しなければなりません（「管理オブジェクトのアクセサメソッド」（45 ページ）を参照）。プリミティブなアクセサメソッド（`primitiveLength`、`setPrimitiveLength:`）は自動的に生成されますが、コンパイラの警告を抑制するため、その宣言をする必要があります。

FoundationコレクションやAppKitビューで使われるオブジェクトについては、通常、スカラ値のインスタンスを生成せず、デフォルトのストレージを使ってプロパティ値を保持するようにしてください。

- スカラ値を保存するために、`NSNumber`オブジェクトラッパを生成し、自動解放後に破棄するようにすれば、CPUやメモリに相応の負荷がかかります。
- Core Dataは、アクセサメソッドがオーバーライドされていなければ、アクセス通知メソッドや変更通知メソッドの呼び出しをインライン化するなど、実行時に最適化を行います。

Core Data自身のストレージを使うようにすれば、スカラ値を直接やり取りする以上に性能上の恩恵が得られます。実際のアプリケーションで、それが本当か疑わしい場合は、性能分析ツールで確認してください。

プロパティはスカラ値として宣言することができます。しかしCore Dataは、スカラ値のアクセサメソッドを動的に生成することができないので、独自に実装する必要があります。モデルで属性lengthをdouble（NSDoubleAttributeType）と指定している場合、インターフェイスファイルで、lengthを次のように宣言します。

```
@property double length;
```

次に、実装ファイルにアクセサを実装します。ここでは、適切なアクセス通知メソッド、変更通知メソッド、プリミティブアクセサを起動してください。プリミティブなアクセサメソッド（primitiveLength、setPrimitiveLength:）は自動的に生成されますが、コンパイラの警告を抑制するため、その宣言をする必要があります（プロパティで宣言可）。

```
@interface MyManagedObject (PrimitiveAccessors)
@property (nonatomic, retain) NSNumber primitiveLength;
@end


- (double)length
{
    [self willAccessValueForKey:@"length"];
    NSNumber *tmpValue = [self primitiveLength];
    [self didAccessValueForKey:@"length"];
    return (tmpValue!=nil) ? [tmpValue doubleValue] : 0.0; // またはnilを表す適切
な表現。
}

- (void)setLength:(double)value
{
    NSNumber* temp = [[NSNumber alloc] initWithDouble: value];
    [self willChangeValueForKey:@"length"];
    [self setPrimitiveLength:temp];
    [self didChangeValueForKey:@"length"];
    [temp release];
}
```

非オブジェクト属性

非標準の属性が、キー値コーディングが対応している構造体（NSPoint、NSSize、NSRect、NSRange）のいずれかであれば、管理オブジェクトモデルでは、型が未定義の一時属性である旨を指定します。エンティティのカスタムクラスの実装には、通常、属性に対応するインスタンス変数を追加します。たとえばboundsという属性があって、これをNSRect構造体で表したい場合、クラスインターフェイスは次のようになります。

```
@interface MyManagedObject : NSManagedObject
{
    NSRect bounds;
}
@property (nonatomic, assign) NSRect bounds;
@end
```

インスタンス変数に属性を保持する場合は、次の例のように、プリミティブな取得/設定アクセサも実装しなければなりません（「プリミティブなアクセサメソッド」（52 ページ）を参照）。

```
@interface MyManagedObject : NSManagedObject
{
    NSRect myBounds;
}
@property (nonatomic, assign) NSRect bounds;
@property (nonatomic, assign) NSRect primitiveBounds;
@end
```

プリミティブなメソッドは、次の例のように、単にインスタンス変数を取得/設定するだけです。キー値監視の変更通知メソッド、アクセス通知メソッドは起動しません。

```
- (NSRect)primitiveBounds
{
    return myBounds;
}
- (void)setPrimitiveBounds:(NSRect)aRect
    myBounds = aRect;
}
```

次にアクセサメソッドを実装しますが、どの戦略であっても、中身はオブジェクト値の場合とほとんど同じです。取得アクセサは遅延型でも事前計算型でも構いません。設定アクセサについても、即時更新型でも遅延更新型でも構いません。以下の節では、遅延型の取得アクセサ、即時更新型の更新アクセサのみ例示します。

取得アクセサ

取得アクセサでは属性値を、管理オブジェクト内部のプライベートなストアから検索します。値が未設定であれば、まだキャッシュしていない可能性があるので、対応する永続値を検索します。それが`nil`でなければ、適当な型に変換し、さらにこの値をキャッシュします。矩形の取得アクセサの例を以下に示します（簡潔にするため、幅は`0`にならないと想定しています。したがって、値が`0`であれば、まだアーカイブから読み込んでいないことになります）。

```
- (NSRect)bounds
{
    [self willAccessValueForKey:@"bounds"];
    NSRect aRect = bounds;
    [self didAccessValueForKey:@"bounds"];
    if (aRect.size.width == 0)
    {
        NSString *boundsAsString = [self boundsAsString];
        if (boundsAsString != nil)
        {
            bounds = NSRectFromString(boundsAsString);
        }
    }
    return bounds;
}
```

設定アクセサ

設定アクセサでは、一時属性と永続属性の両方に、同時に値を設定します。非標準型の値は、標準型に変換した上で、永続値として設定します。このとき、キー値監視の変更通知メソッドを起動して、管理オブジェクトを監視しているオブジェクト（管理オブジェクトコンテキストを含む）に、修正した旨を通知しなければなりません。矩形の設定アクセサの例を以下に示します。

```
- (void)setBounds:(NSRect)aRect
{
    [self willChangeValueForKey:@"bounds"];
    bounds = aRect;
    [self didChangeValueForKey:@"bounds"];
    NSString *rectAsString = NSStringFromRect(aRect);
    [self setValue:rectAsString forKey:@"boundsAsString"];  }
```

型チェック

非標準型の属性を定義する場合、値を表現するクラスの名前も、`setAttributeValueClassName:`で指定できます。

値クラス名の設定は、コードでしかできません。管理オブジェクトモデルを修正して、非標準型の属性（`favoriteColor`）を表現する値クラスの名前を設定する例を示します。この例では、値はカスタムクラス`MyColor`のインスタンスで表されます。

```
myManagedObjectModel = <#Get a managed object context#>;

NSEntityDescription *employeeEntity =
    [[myManagedObjectModel entitiesByName] objectForKey:@"Employee"];
NSAttributeDescription *favoriteColorAttribute =
    [[employeeEntity attributesByName] objectForKey:@"favoriteColor"];

// 属性を表現する値クラスをMyColorと設定
[favoriteColorAttribute setAttributeValueClassName:@"MyColor"];
```

属性の値クラスは、実行時に、実際に存在しなければなりません。クラス名の綴りを間違っていても（たとえば`MyColor`ではなく`MyColour`と記述）、チェックには成功してしまうので注意してください。

Core Dataは、属性値として設定された値のクラスを調べ、誤ったクラスのインスタンスであれば例外を投げます。

```
Employee *newEmployee =
        [NSEntityDescription insertNewObjectForEntityForName:@"Employee"
            inManagedObjectContext:aManagedObjectContext];
newEmployee.favoriteColor = [NSColor redColor]; // ここで例外が発生。
```

管理オブジェクトの検証

検証には、プロパティレベルとプロパティをまたがるものの2種類があります。個々の値の正当性はプロパティレベル、値の組み合わせの正当性はプロパティをまたがる検証で確認します。

Core Dataの検証機能

Cocoaにはモデル値を検証するための基本的な機能が組み込まれています（『*Model Object Implementation Guide*』の「モデルオブジェクトの検証」を参照）。しかしこのアプローチでは、検証したい制約それぞれについて、コードを記述しなければなりません。これに対し、Core Dataでは、検証ロジックを管理オブジェクトモデルに組み込めるようになっています。数値属性、日付属性には、最大値と最小値を指定できます。文字列属性には、長さの最大値、最小値に加え、合致しなければならない正規表現も指定できます。さらに、関係についても、必須である、ある個数を超えてはならない、などといった制約を指定できます。このように、属性値の一般的な制約については、コードを記述しなくても指定できるのです。

個々のプロパティの検証方法をカスタマイズする場合、`NSKeyValueCoding`プロトコルで定義された、標準的な検証メソッドが使えます（「プロパティレベルの検証」（99ページ）を参照）。Core Dataでは、関係や、プロパティをまたがる値も検証できるよう拡張されています。詳しくは「プロパティをまたがる検証」（101 ページ）を参照してください。

重要なのは、*どのように*検証するか、はモデルで判断する事項、いつ検証するか、はユーザインターフェイスやコントローラレベルで判断する事項である、ということです（たとえばテキストフィールドの値については、入力後「直ちに検証する」設定になっていることが多いでしょう）。管理オブジェクトやオブジェクトグラフの不整合は、さまざまな時点で起こりえます。

メモリ上のオブジェクトに、一時的に不整合が生じても、それ自体は問題ありません。Core Dataが検証制約を適用するのは、「保存」操作の時点か、または明示的に要求されたときだけです（検証メソッドはいつでも直接起動可）。確かに、変更が施された時点で即座に検証し、エラーを報告する方が有用な状況もあります。これにより、最終的に保存する時点になって膨大な数のエラーが表示される、という事態を回避できるからです。しかしながら、管理オブジェクトが常に妥当な状態でなければならないとすれば、エンドユーザがあるワークフローに従うよう強いられるなど、問題が生じます。「作業領域」に相当する、管理オブジェクトコンテキストというものを使うのも、このためです。通常、管理オブジェクトを作業領域に持ってきてさまざまな編集を行い、最後にまとめて変更をコミットするか、破棄して元の状態に戻すことになります。

プロパティレベルの検証

`NSKeyValueCoding`プロトコルでは、全般的な検証機能を提供するメソッドとして、`validateValue:forKey:error:`を指定しています（アクセサメソッドの機能を`valueForKey:`で提供する、という規約と同様の考え方）。

制約に加え、何らかの処理ロジックを管理オブジェクトモデルに実装する場合、`validateValue:forKey:error:`をオーバーロードしてはいけません。代わりに、`validate<Key>:error:`という形のメソッドを実装してください。

重要： カスタム検証メソッドを実装しても、通常、これを直接呼び出してはなりません。適切なキーを指定して、`validateValue:forKey:error:`を呼び出すようにしてください。こうすれば、管理オブジェクトモデルで定義した制約も適用されます。

メソッドの実装では、新たに設定されようとしている値を検査し、制約を満たさなければ`NO`を返します。error引数が`null`でなければ、次のコード例のように、エラーの内容を記述した`NSError`オブジェクトも生成します。

```
-(BOOL)validateAge:(id *)ioValue error:(NSError **)outError {
    if (*ioValue == nil) {
        // setNilValueForKeyでこれをトラップ? 値が0のNSNumberを生成?
        return YES;
    }
    if ([*ioValue floatValue] <= 0.0) {
        if (outError != NULL) {
            NSString *errorStr = NSLocalizedStringFromTable(
                @"Age must greater than zero", @"Employee",
                @"validation: zero age error");
            NSDictionary *userInfoDict = [NSDictionary
dictionaryWithObject:errorStr
                forKey:NSLocalizedDescriptionKey];
            NSError *error = [[[NSError alloc]
initWithDomain:EMPLOYEE_ERROR_DOMAIN
                code:PERSON_INVALID_AGE_CODE
                userInfo:userInfoDict] autorelease];
            *outError = error;
        }
        return NO;
    }
    else {
        return YES;
    }
    // . . .
```

入力値はオブジェクト参照を指すポインタです（`id *`）。したがって、この入力値を変更することもできてしまいます。しかし、メモリ管理上重大な問題が生じうるので、これはお勧めできません（『*Key-Value Coding Programming Guide*』の「キー値の検証」を参照）。さらに、プロパティの検証を行うカスタムメソッドでは、`validateValue:forKey:error:`も呼び出してはいけません。実行時に`validateValue:forKey:error:`が起動された時点で、無限ループに陥るからです。

`validate<Key>:error:`メソッドでは入力値を変更することも可能ですが、正しくない、あるいは選択肢にない値の場合に限るべきです。値を変更すればオブジェクトやコンテキストは「汚染された」状態になるため、Core Dataが再び検証しようとする可能性があります。したがって、検証メソッド内で強制的に値を修正すると、無限ループに陥る恐れがあるのです。同様に、検証メソッドや`willSave`メソッドを、状態変化や副作用を生じるように実装する場合にも注意が必要です。Core Dataは安定した状態になるまで、変更を何度でも検証しようとするからです。

プロパティをまたがる検証

オブジェクトの個々の属性値がすべて妥当であっても、その組み合わせが正しくない可能性は残ります。たとえば、あるアプリケーションで、人物の年齢および運転免許証の有無を格納するとしましょう。Personオブジェクトで、age属性の値が12であること、hasDrivingLicense属性の値がYESであることは、個別には妥当なのですが、（少なくとも大部分の国で）この組み合わせが妥当とは言えません。

NSManagedObjectでは、更新、挿入、削除の際に、このような検証を行えるようになっています。validateFor…という形のメソッド（validateForUpdate:など）を使います。プロパティをまたがる検証用のカスタムメソッドを実装する場合、その先頭でスーパークラスの実装を呼び出して、個々のプロパティの検証メソッドも起動するようにしなければなりません。スーパークラス側の検証メソッドで不合格になった（妥当でない属性値が見つかった）場合、次の2通りの処理が考えられます。

1. スーパークラス側で生成されたエラー情報を添えてNOを返す。
2. 検証を続行し、値の組み合わせが妥当かどうか確認する。

続行する場合、検証ロジックで参照する個々の値が妥当ではないため、組み合わせの検証でエラーになる可能性があります（たとえば、ある属性が正値であるはずなのに実際には0である場合、組み合わせの検証コードに除算があればエラーになる）。また、組み合わせが妥当でない値が見つかった場合、個別の値が妥当でなかったことと併せ、「複数のエラー」があるものとして報告しなければなりません（「複数の検証エラーの併合」（103 ページ）を参照）。

プロパティをまたがる検証メソッドの実装例を以下に示します。Personエンティティには、birthdayおよびhasDrivingLicenseという2つの属性があるとします。検証するべき制約は、16歳未満ならば運転免許証を所持できない、というものです。この制約を、validateForInsert:とvalidateForUpdate:の両方で検証しているので、検証ロジック自体は2つのメソッドに分かれています。

リスト 1　Personエンティティのプロパティをまたがる検証

```
- (BOOL)validateForInsert:(NSError **)error
{
    BOOL propertiesValid = [super validateForInsert:error];
    // 個々の値が妥当でなければここで終了する、という実装も可能
    BOOL consistencyValid = [self validateConsistency:error];
    return (propertiesValid && consistencyValid);
}

- (BOOL)validateForUpdate:(NSError **)error
{
    BOOL propertiesValid = [super validateForUpdate:error];
    // 個々の値が妥当でなければここで終了する、という実装も可能
    BOOL consistencyValid = [self validateConsistency:error];
    return (propertiesValid && consistencyValid);
}


- (BOOL)validateConsistency:(NSError **)error
{
    static     NSCalendar *gregorianCalendar;
```

```
    BOOL valid = YES;
    NSDate *myBirthday = [self birthday];

    if ((myBirthday != nil) && ([[self hasDrivingLicense] boolValue] == YES))
{

        if (gregorianCalendar == nil) {
            gregorianCalendar = [[NSCalendar alloc]
initWithCalendarIdentifier:NSGregorianCalendar];
        }
        NSDateComponents *components = [gregorianCalendar
components:NSYearCalendarUnit
                                                                fromDate:myBirthday
                                                                  toDate:[NSDate
date]
                                                                 options:0];
        int years = [components year];

        if (years < 16) {

            valid = NO;

            // 要求がなければエラー通知オブジェクトを生成しない
            if (error != NULL) {

                NSBundle *myBundle = [NSBundle bundleForClass:[self class]];
                NSString *drivingAgeErrorString = [myBundle
localizedStringForKey:@"TooYoungToDriveError"
                                    value:@"Person is too young to have a driving
 license."
                                    table:@"PersonErrorStrings"];

              NSMutableDictionary *userInfo = [NSMutableDictionary dictionary];
                [userInfo setObject:drivingAgeErrorString
forKey:NSLocalizedFailureReasonErrorKey];
                [userInfo setObject:self forKey:NSValidationObjectErrorKey];

              NSError *drivingAgeError = [NSError errorWithDomain:PERSON_DOMAIN

code:NSManagedObjectValidationError
                                                              userInfo:userInfo];

                // 以前にエラーが出ていなければ、新たに生成したエラー通知オブジェクトを返す
                if (*error == nil) {
                    *error = drivingAgeError;
                }
                // 以前にもエラーが出ていれば、その時のエラー通知オブジェクトに併合する
                else {
                    *error = [self errorFromOriginalError:*error
error:drivingAgeError];
                }
            }
        }
    }
    return valid;
}
```

複数の検証エラーの併合

1回の操作で複数の検証エラーが見つかった場合は、「複数のエラー」が見つかった旨のオブジェクト、すなわち、コードを`NSValidationMultipleErrorsError`とした`NSError`オブジェクトを生成して返します。個別のエラー情報を配列にまとめ、`NSDetailedErrorsKey`というキーで、`NSError`オブジェクトのユーザ情報辞書に追加します。スーパークラスの検証メソッドでエラーが見つかった場合にも、この方法で処理します。実行するテストの数にもよりますが、既存の`NSError`オブジェクト（この時点ですでに「複数のエラー」である可能性もある）に、新たに見つかったエラー情報を追加して返すメソッドを定義しておくと便利でしょう。

2つのエラー情報を「複数のエラー」として併合するメソッドの実装例を以下に示します。組み合わせる処理の実装は、併合元が「複数のエラー」であるか否かによって違います。

リスト 2　　2つのエラー情報を「複数のエラー」として併合するメソッド

```
- (NSError *)errorFromOriginalError:(NSError *)originalError error:(NSError
*)secondError
{
    NSMutableDictionary *userInfo = [NSMutableDictionary dictionary];
    NSMutableArray *errors = [NSMutableArray arrayWithObject:secondError];

    if ([originalError code] == NSValidationMultipleErrorsError) {

        [userInfo addEntriesFromDictionary:[originalError userInfo]];
       [errors addObjectsFromArray:[userInfo objectForKey:NSDetailedErrorsKey]];
    }
    else {
        [errors addObject:originalError];
    }

    [userInfo setObject:errors forKey:NSDetailedErrorsKey];

    return [NSError errorWithDomain:NSCocoaErrorDomain
                               code:NSValidationMultipleErrorsError
                           userInfo:userInfo];
}
```

フォールティングと一意化

フォールティングは、アプリケーションのメモリ消費量を削減するための、Core Dataのメカニズムです。これに関連する、一意化という機能は、管理オブジェクトコンテキストの範囲内で、同じレコードを表現する管理オブジェクトが複数にならないようにする働きがあります。

フォールティングによるオブジェクトグラフの大きさ制限

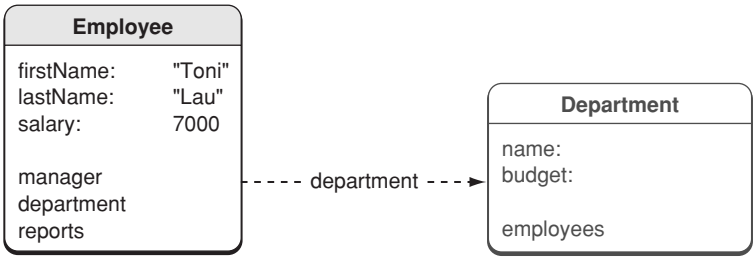
フォールティングにより、アプリケーションのメモリ消費量を削減できます。「フォールト」とは、まだ完全に実体化していない管理オブジェクトや、関係のコレクションオブジェクトを表す、プレースホルダのことです。

- 管理オブジェクトのフォールトは、あるクラスのインスタンスであって、その永続変数がまだ初期化されていないものです。
- 一方、関係のフォールトは、コレクションクラスのサブクラスとして、関係を表現できるように実装されています。

この仕組みにより、オブジェクトグラフには、「ここから先は実体化されていない」という「壁」が生じます。フォールトはまだ実体化していないので、管理オブジェクトのフォールトはメモリ消費量がわずかであり、また、これと関係で結ばれた管理オブジェクトは、まったくメモリを消費しません。

例として、従業員に関する詳細情報を1人分ずつフェッチし、編集するというアプリケーションを考えてみましょう。従業員には、関係先として上長（manager）、部署（department）などがあり、関係先オブジェクトもさらに別の関係によってほかのオブジェクトとつながっています。Employeeオブジェクトを1つ、永続ストアから検索したばかりの段階では、manager、department、reportsの関係は「フォールト」として表されています。図 1に、ある従業員の関係「department」がフォールトで表されている様子を示します。

図 1　department関係がフォールトで表されている様子



このフォールトはDepartmentクラスのインスタンスですが、まだ実体化されていません。すなわち、永続インスタンス変数が未設定です。したがって、departmentオブジェクトがほとんどメモリを消費しないだけでなく、ここからさらにemployees関係をたどる必要もありません。これに対し、オブジェクトグラフを完全に実体化してからでないと従業員の属性を編集できないとすれば、会社組織全体を表すオブジェクトをすべて生成しなければならなくなってしまいます。

フォールトの処理は「透過的」です。すなわち、実体化するために、明示的にフェッチする必要はありません。ある時点で、フォールト状態のオブジェクトの永続属性がアクセスされれば、Core Dataは自動的に該当するデータを検索し、オブジェクトを初期化します（フォールトが発動することのないメソッドの一覧が『*NSManagedObject Class Reference*』にあります）。このプロセスを一般に、「フォールトが発動する」と言います。たとえば、あるDepartmentオブジェクトに名前を取得するメッセージを送ると、フォールトが発動します。するとCore Dataは、自動的にフェッチを実行し、当該オブジェクトの属性をすべて検索します。

フォールトの発動

Core Dataでは、必要が生じると（フォールトの永続属性がアクセスされると）、自動的にフォールトが発動するようになっています。しかし、個々のフォールトがばらばらに発動するのでは、効率的でないかもしれません。そこで、永続ストアからデータを取得する、よりよい戦略があります（「SQLiteストアにおけるフォールトの一括実体化と事前フェッチ」（137ページ）を参照）。フォールトを実体化し、関係をたどる処理の効率を改善する方法については、「管理オブジェクトのフェッチ」（136 ページ）を参照してください。

フォールトが発動しても、該当するデータがキャッシュ上にあれば、ストアを検索することはありません。キャッシュがヒットすれば、効率的に管理オブジェクトを実体化できます。これは基本的に、通常の管理オブジェクトのインスタンス生成と同じです。データがキャッシュ上になければ、Core Dataは自動的に、該当するオブジェクトのフェッチを実行します。すなわち、永続ストアを検索してデータをフェッチし、メモリ上にキャッシュすることになります。

以上からも分かるように、あるオブジェクトがフォールトか否かと、データがストアから検索されるか否かは、別の概念です。オブジェクトがフォールトか否かは、該当する管理オブジェクトの属性がすべて、すぐに使える状態になっているかどうか、を表すにすぎません。また、これは`isFault`メッセージを送ることにより、発動させることなく判別できます。戻り値が`NO`であれば、データはメモリ上にあります。もっとも、戻り値が`YES`であったとしても、データがメモリ上にないとは*限りません*。これはキャッシュに関係するさまざまな要因によって決まるからです。

オブジェクトのフォールト化

いったん実体化したオブジェクトをフォールト化することが、役に立つ場合もあります。プロパティ値を確実に最新状態にする（「データが最新であることの保証」（72 ページ）を参照）だけでなく、オブジェクトグラフのうち当面必要としない部分を刈り込む（「メモリ消費量のオーバーヘッド削減」（139ページ）を参照）という効果もあるからです。実際、管理オブジェクトをフォールト化すると、不要なメモリが解放されてメモリ上のプロパティ値が`nil`になるほか、保持されていた関係先オブジェクトも解放されます。

実体化されたオブジェクトのフォールト化には、`refreshObject:mergeChanges:`メソッドを使います。*`mergeChanges`*引数として`NO`を指定する場合、オブジェクトの関係を変更していないことが確実でなければなりません。変更があれば、コンテキストを保存する際、永続ストアに参照整合性の問題が起こります。

オブジェクトをフォールト化すると、`didTurnIntoFault`メッセージが送信されます。さまざまな「ハウスキーピング」処理を行う、カスタムメソッド`didTurnIntoFault`を実装してもよいでしょう（その実例は「データが最新であることの保証」（72 ページ）を参照）。

注： Core Dataでは混乱を避けるため、実体化（faulting）の逆の操作について、「逆実体化」（*unfaulting*）という用語は使わないようにしています。仮想メモリに関して、ページフォールトを「unfaulting」する、ということはありません。ページフォールトとは、（OSが）積極的に引き起こすものではなく、（メモリ管理ユニットというハードウェアにより）引き起こされるもの、発生するものです。もちろん、メモリを解放してカーネルに返却するためには、さまざまな手段が用意されています（`vm_deallocate`、`munmap`、`sbrk`などの関数）。Core Dataではこれを、「オブジェクトをフォールト化する」と表現しています。

フォールトとKVO通知

Core Dataがオブジェクトをフォールト化すると、当該オブジェクトのプロパティに関する、KVO（キー値監視）変更通知（『*Key-Value Observing Programming Guide*』を参照）が送られます。フォールト化されたオブジェクトのプロパティを監視していて、その後これが実体化された場合、プロパティ値が実際には変わっていなくても、変更通知を受け取ることになります。

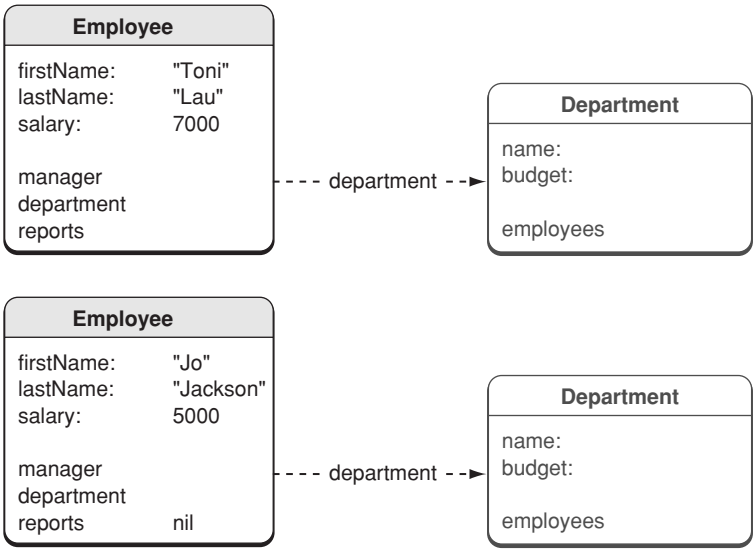
実体化の処理は透過的に行われるため、外見上の値は変わっていないのですが、メモリ上の該当する領域を見ると、バイト並びが変化しています。そしてKVO（キー値監視）機構が働くためには、ポインタ値として比較したバイト並びが変化していれば通知を出す、という仕組みになっている必要があります。KVOはこの通知を利用して、キーパスや依存オブジェクトをまたがる変更を追跡します。

同一コンテキスト、同一レコードに対応する管理オブジェクトの一意化

Core Dataは、同じ管理オブジェクトコンテキストの範囲内で、永続ストアの同じレコードに対応する管理オブジェクトが、いくつも生成されることのないようにしています。この処理を**一意化**と言います。これをしないと、あるレコードに対応するオブジェクトが、同一コンテキスト内に複数存在することになってしまいます。

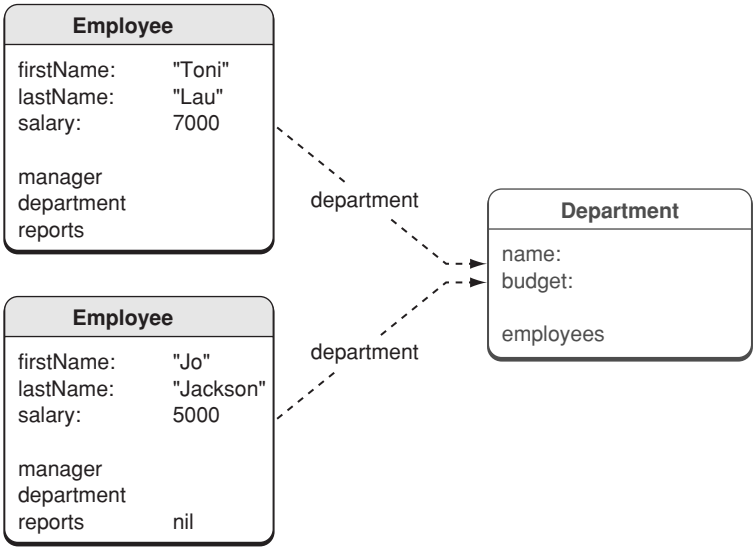
たとえば図2のような状況を考えてみましょう。2人の従業員が*同じ管理オブジェクトコンテキスト*内にフェッチされています。どちらにもdepartmentとの関係がありますが、現時点ではフォールトになっています。

図 2 2つのdepartmentオブジェクトがそれぞれフォールトになっている様子



2人の従業員はそれぞれ、ある部署に所属しています。どの部署に属しているか問い合わせる（アクセスする）と、フォールトは実体化して通常のオブジェクトになり、メモリ上には2つのDepartmentオブジェクトが生成されることになります。しかし、実際には2人とも同じ部署（たとえば「マーケティング部門」）に属していたとすれば、（同じ管理オブジェクトコンテキストの範囲内では）1つだけ、マーケティング部門を表すオブジェクトを生成するようになっています。その結果、図3のように、「department」関係は同じフォールトを参照するようになります。

図 3 一意化されたフォールト：2人の従業員が同じ部署に属している場合



Core Dataに一意化の機能がなかったとすれば、全従業員をフェッチしてどの部署に属しているか問い合わせた（したがって該当するフォールトが発動した）場合、そのたびに新しいDepartmentオブジェクトが生成されてしまいます。同じ部署を表すオブジェクトが複数あると、データが食い違い、不整合が生じる恐れがあります。コンテキストを保存する際にも、どれを保存すればよいのか判断できません。

これに限らず、同じコンテキスト内の管理オブジェクトで、「マーケティング部門」のDepartmentオブジェクトを参照しているものは、すべて同じインスタンスを参照しています。つまり、「マーケティング部門」に関するデータはひとつしかありません。*これが現時点ではフォールトであっても、事情は同じです。*

注： 以上に述べた内容は、同じ管理オブジェクトコンテキスト内に閉じた話です。管理オブジェクトコンテキストが複数あれば、これはそれぞれ独立したデータビューを表します。同じ従業員を異なるコンテキストにフェッチした場合、この従業員も、関係で結ばれたDepartmentオブジェクトも、メモリ上ではすべて別のオブジェクトとして表されます。異なるコンテキストにあるオブジェクトは、データが違っていても*構いません*。保存の際にこの違いを検出し、解決するのは、まさにCore Dataアーキテクチャの役割です。

永続ストアの使い方

この章では、永続ストアを生成し、ある型のストアを別の型に移行し、ストアのメタデータを管理する方法を解説します。永続ストアの型と、それぞれの型の違いおよび動作の設定手順については、「永続ストアの特徴」（125 ページ）を参照してください。

ストアの生成とアクセス方法

ストアへのアクセスは、`NSPersistentStoreCoordinator`のインスタンスを介して行います。ストアを格納しているファイルに、直接アクセスする必要はありません。永続ストアコーディネータから、ディスク上にある個々のストアを表すオブジェクトを検索できます。Core Dataには、永続ストアを表す、`NSPersistentStore`というクラスが用意されています。

ストアの生成には永続ストアコーディネータを使います。そのストアの型を指定するほか、必要に応じ、コーディネータに関連付ける管理オブジェクトモデル、および、メモリ内ストア以外の場合はその場所を指定できます。読み込み専用のXMLストアを生成するコード例を以下に示します。

```
NSManagedObjectContext *moc = <#Get a context#>;
NSPersistentStoreCoordinator *psc = [moc persistentStoreCoordinator];
NSError *error = nil;
NSDictionary *options =
    [NSDictionary dictionaryWithObject:[NSNumber numberWithBool:1]
                    forKey:NSReadOnlyPersistentStoreOption];

NSPersistentStore *roStore =
    [psc addPersistentStoreWithType:NSXMLStoreType
                    configuration:nil URL:url
                    options:options error:&error];
```

コーディネータからストアオブジェクトを検索するには、`persistentStoreForURL:`メソッドを使います。次のコード例のように、特定のストアのみを対象とするよう、フェッチ要求を制限できます。

```
NSPersistentStoreCoordinator *psc = <#Get a coordinator#>;
NSURL *myURL = <#A URL identifying a store#>;
NSPersistentStore *myStore = [psc persistentStoreForURL:myURL];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setAffectedStores:[NSArray arrayWithObject:myStore]];
```

ストアの型や場所の変更

ある型のストアから別の型に、あるいは、ある場所にあるストアを別の場所に（たとえば「名前を指定して保存」コマンドの実装）移行するためには、`NSPersistentStoreCoordinator`の`migratePersistentStore:toURL:options:withType:error:`メソッドを使います。このメソッド

を実行すると、移行元のストアはコーディネータから削除されるので、使えなくなります。このメソッドの使用例を以下に示します。ある場所にあるストアを別の場所に移行しています。また、元のストアがXML型であれば、SQLite型に変換します。

```
NSPersistentStoreCoordinator *psc = [aManagedObjectContext persistentStoreCoordinator];
NSURL *oldURL = <#URL identifying the location of the current store#>;
NSURL *newURL = <#URL identifying the location of the new store#>;
NSError *error = nil;
NSPersistentStore *xmlStore = [psc persistentStoreForURL:oldURL];
NSPersistentStore *sqLiteStore = [psc migratePersistentStore:xmlStore
    toURL:newURL
    options:nil
    withType:NSSQLiteStoreType
    error:&error];
```

ストアの移行は次の手順で行います。

1. 一時的な永続スタックを生成する
2. 旧ストアと新ストアをマウントする
3. 旧ストアからオブジェクトをすべて読み込む
4. オブジェクトを新ストアに移行する

   オブジェクトに一時IDを与えた上で、新ストアに割り当てます。新ストアでは、新たに割り当てられたオブジェクトを保存します（外部リポジトリにコミット）。

   Core Dataは次に、ほかのスタックに、オブジェクトIDが変更された（旧ストアから新ストアに）旨を通知します。移行後も正しく動作するためにはこれが必要です。
5. 旧ストアをマウント解除する
6. 新ストアに戻る

次のような場合はエラーになります。

- メソッドに指定したパラメータが正しくない
- 新しいストアを追加できない
- 旧ストアを削除できない

追加/削除できない場合のエラーは、`addPersistentStore:`や`removePersistentStore:`を直接呼び出した場合と同じです。ストアを追加または削除する際にエラーが発生した場合、永続スタックの整合性が崩れている可能性が高いので、例外として扱うべきでしょう。

移行そのものの過程に問題があると、エラーではなく例外になります。この場合、完全に移行前の状態に戻るので、修復処理は必要ありません。例外記述を調べることにより、問題点を判断できます。考えられるエラーの原因は、「ディスク容量不足」や「操作権限の欠如」といったものから、「SQLiteストアの破損」や「Core Dataがストアをまたがる関係に対応していない」まで、多岐にわたります。

メタデータの設定:ストアに関する追加情報、高度な検索インデックス（スポットライト）

ストアの**メタデータ**とは、ストアに関する追加情報のうち、ストア内のどのエンティティにも直接には関連付けられていないもののことです。

メタデータは辞書の形で表します。ストアの型およびUUIDは、自動的にメタデータとしてキーと値の組が登録されます。ほかにもキーを追加してメタデータを登録できます。実際のアプリケーションに応じたカスタムキーのほか、高度な検索インデックス「スポットライト」を実装するための`kMDItemKeywords`など、標準で用意されているキーもあります（「スポットライト」の実装には相応するインポータの記述も必要）。

メタデータとして記述する情報は、慎重に決めなければなりません。まず、「スポットライト」なので、メタデータの容量には制約があります。次に、メタデータ内の全文書を複製しても、恐らくそれは役に立ちません。しかしながら、ストア内のオブジェクトを一意に識別するURLを生成することは可能です（`URIRepresentation`を使用）。このURLをメタデータとして記述すれば有用かもしれません。

メタデータの取得

ストアのメタデータを取得する手段は2通りあります。

1. 永続ストアのインスタンスが与えられれば、`NSPersistentStoreCoordinator`のインスタンスメソッド`metadataForPersistentStore:`で、メタデータを取得できます。

2. `NSPersistentStoreCoordinator`のクラスメソッド`metadataForPersistentStoreOfType:URL:error:`でもメタデータを取得できます。永続スタックを生成する、というオーバーヘッドがありません。

この2つには重要な違いがあります。インスタンスメソッド`metadataForPersistentStore:`の戻り値は、その時点でプログラムで使っている状態のメタデータです。したがって、ストアを最後に保存して以降に修正が施されていれば、それを反映したものになります。一方、クラスメソッド`metadataForPersistentStoreOfType:URL:error:`の戻り値は、その時点でストア自体に格納されているメタデータです。ストアに変更を施してまだ保存していなければ、この戻り値は最新の状態を表していないことになります。

メタデータの設定

ストアにメタデータを設定する手段は2通りあります。

1. 永続ストアのインスタンスが与えられれば、`NSPersistentStoreCoordinator`のインスタンスメソッド`setMetadata:forPersistentStore:`で、メタデータを設定できます。

2. `NSPersistentStoreCoordinator`のクラスメソッド`setMetadata:forPersistentStoreOfType:URL:error:`でもメタデータを設定できます。永続スタックを生成する、というオーバーヘッドがありません。

この2つにも、同様に、重要な違いがあります。setMetadata:forPersistentStore:を使う場合、事前に（管理オブジェクトコンテキストを介して）ストアを保存する必要があります。一方、setMetadata:forPersistentStoreOfType:URL:error:を使えば、メタデータは即座に更新されます（ファイルの最終更新日時も変わります）。Mac OS X上でNSPersistentDocumentを使う場合、この違いはより顕著に表れます。永続ストア上でアクティブに作業中に（すなわち、まだ保存していない変更事項がある状態で）、メタデータをsetMetadata:forPersistentStoreOfType:URL:error:で更新すれば、文書を保存する時点で、「This document's file has been changed by another application since you opened or saved it.」という警告が現れます。これを回避したければ、代わりにsetMetadata:forPersistentStore:を使わなければなりません。文書の永続ストア（インスタンス）を見つけるためには、通常、永続ストアコーディネータに対してその永続ストア（persistentStores）を問い合わせ、返された配列の先頭要素を使います。

Core DataはNSStoreTypeおよびNSStoreUUIDの値を管理しているので、次のコード例のように、既存のメタデータのミュータブルな複製を作ってから、独自のキーと値を設定するようにしなければなりません。

```
NSError *error = nil;
NSURL *storeURL = <#URL identifying the location of the store#>;

NSDictionary *metadata =
    [NSPersistentStore metadataForPersistentStoreWithURL:storeURL error:&error]
if (metadata == nil) {
    /* エラーを処理する */
}
else {
    NSMutableDictionary *newMetadata =
            [[metadata mutableCopy] autorelease];
    [newMetadata setObject:[NSArray arrayWithObject:@"MyKeyWord"]
            forKey:(NSString *)kMDItemKeywords];
    // 必要に応じて追加のキーと値の組を設定。
    [NSPersistentStore setMetadata:newMetadata
                        forPersistentStoreWithURL:storeURL
                        error:&error];
}
```

Core DataとCocoaバインディング

オブジェクトのプロパティ値に変更を施せば、その影響はユーザインターフェイス上にも現れるはずです。このとき、同じプロパティに対応するユーザインターフェイス要素は、同期して変化しなければなりません。Cocoaバインディングはこのような同期のための制御階層を提供するものですが、Core Dataフレームワークがモデルの側に着目するのに対し、Cocoaバインディングはユーザインターフェイスの側に着目する仕組みになっています。多くの場合、Cocoaバインディングを使えば、ユーザインターフェイスに関するプロパティの同期を保つのが容易になります。Core Dataフレームワークは、Cocoaバインディングとシームレスに併用し、その使い勝手を改善できるように設計されています。

iOS： Cocoaバインディングは、iOSでは使えません。

CocoaバインディングとCore Dataの概念は、多くの面で直交している、と言えるでしょう。一般にCocoaバインディングは、管理オブジェクトを、ほかのCocoaモデルオブジェクトとまったく同じように扱います。また、永続ストアからオブジェクトをフェッチするために使うのと同じ述語オブジェクトや整列記述子が、メモリ中のオブジェクトを（たとえば表の形で表示するために）しぼり込み、整列するためにも使えます。したがって、アプリケーション全体で、一貫したAPIを利用できるのです。しかし、若干ですが、設定や操作には違いがあります（多くは自明の違いです）。

この章で説明する問題以外にも、Core DataとCocoaバインディングの組み合わせより問題が生じうる箇所が若干あります。「Core Dataのトラブルシューティング」（143 ページ）の、特に次の項を参照してください。

- 「関係の設定ミューテータ（カスタムメソッド）が配列コントローラから呼び出されない」（150 ページ）
- 「Nibファイルの読み込み後、オブジェクトコントローラの内容にアクセスできない」（150ページ）
- 「テーブルビューまたはアウトラインビューが、NSArrayControllerまたはNSTreeControllerオブジェクトをバインドしたとき、最新の内容に維持できない」（151 ページ）

こういった例外はありますが、『*Cocoa Bindings Programming Topics*』に説明されている事項はすべて、Core Dataベースのアプリケーションに適用されます。したがって、バインディングの設定やデバッグは、Core Dataを使っている場合であっても、そうでない場合と同じように行うとよいでしょう。

コントローラに関する追加機能

Cocoaバインディングに対してCore Dataが追加する主な機能は、コントローラオブジェクト（`NSObjectController`、`NSArrayController`など）の設定に関するものです。Core Dataはこのクラスに次のような機能を追加しています。

- 管理オブジェクトコンテキストの参照。これはフェッチ、挿入、削除の操作すべてに使われます。

  コントローラの内容が管理オブジェクト、またはそのコレクションである場合、コントローラには、管理オブジェクトコンテキストをバインドまたは設定しなければなりません。

- エンティティ名。新規オブジェクトを生成する際、そのクラスの代わりに使います。

- フェッチ述語の参照。内容が直接設定されていない場合に、何をフェッチして設定するかを制約します。

- 「Deletes Objects On Remove」というバインディングオプション。（内容が関係にバインドされている場合に、）オブジェクトをコントローラから削除したとき、関係を切断するだけでなく、当該オブジェクト自身も削除する、という指定です。

「Automatically Prepares Content」フラグ

コントローラの「automatically prepares content」フラグ（たとえば`setAutomaticallyPreparesContent:`を参照）がオンになっていれば、コントローラの当初の内容が、対応する管理オブジェクトコンテキストから、その時点でのコントローラのフェッチ述語を使ってフェッチされます。重要なのは、コントローラのフェッチが、（Nibファイルを読み込んで）管理オブジェクトコンテキストを設定した後で、遅延実行されることです。すなわち、`awakeFromNib`および`windowControllerDidLoadNib:`の後で実行されるのです。そのため、オブジェクトコントローラの内容をこのいずれかのメソッドで操作しようとしても、その時点ではコントローラの内容が`nil`であるために、問題が生じる場合があります。これを回避するためには、`fetchWithRequest:merge:error:`を使って「手動で」フェッチしなければなりません。次のコード例のように、フェッチ要求の引数として`nil`を渡すことにより、デフォルトのフェッチ要求を行います。

```
- (void)windowControllerDidLoadNib:(NSWindowController *) windowController
{
    [super windowControllerDidLoadNib:windowController];

    NSError *error = nil;
    BOOL ok = [arrayController fetchWithRequest:nil merge:NO error:&error];
    // ...
```

エンティティの継承

フェッチ要求するエンティティとしてスーパーエンティティを指定すると、それを含むサブエンティティすべてのうち、条件に合致するインスタンスが返されます（「フェッチ処理とエンティティの継承」（64 ページ）を参照）。同様に、コントローラのエンティティとしてスーパーエンティティを指定しても、それを含むサブエンティティすべてのうち、条件に合致するインスタンスをフェッチできます。スーパーとなる抽象エンティティを指定すれば、サブの具象エンティティで条件に合致するものをフェッチできます。

対多関係を対象としぼり込み述語

検索フィールドにしぼり込み述語を設定することにより、対多関係で結ばれた先を基準として、配列コントローラの内容をしぼり込みたい、という状況もあります。対多関係を検索する場合、述語にANYまたはALLを指定する必要があります。たとえば、「Matthew」という名前の従業員が1人でも所属する部署（Department）をフェッチしたい場合、次のように、ANY演算子を使います。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"ANY employees.firstName like 'Matthew'"];
```

フィールドの述語バインディングも同様です。

```
ANY employees.firstName like $value
```

注： contains演算子を用いた、たとえば「ANY employees.firstName contains 'Matthew'」という述語は使えません。contains演算子はANY演算子と組み合わせることができないからです。

前方一致や後方一致による照合は、より複雑になります。たとえば、「Matt」、「Matthew」、「Mattie」など、「Matt」で始まる名前の従業員がいる部署を検索する、というような状況です。これは次のように、ワールドカードで照合することになるでしょう。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"ANY employees.firstName like 'Matt*'"];
```

しかし検索フィールドの述語バインディングでは、この構文が*使えません*。

```
// 想定通りの結果にはならない
ANY employees.firstName like '$value*'
```

その理由は『*Predicate Programming Guide*』に説明があります。述語中に引用符があると、その範囲では変数置換が起こらないのです。そのため、次のように、ワイルドカードを先に置換しなければなりません。

```
NSString *value = @"Matt";
NSString *wildcardedString = [NSString stringWithFormat:@"%@*", value];
[[NSPredicate predicateWithFormat:@"ANY employees.firstName like %@",
wildcardedString];
```

以上から分かるように、このような動作を実現しようとすれば、若干のコードを記述する必要があります。

注： 検索フィールドの述語バインディングによりしぼり込みを行うと、その結果はワイルドカード文字に関して整合性が取れていないことがあります。これはNSArrayControllerのバグによるものです。回避するためには、NSArrayControllerのサブクラスを定義し、arrangeObjects:をオーバーロードしてください。その中身は、superの実装を呼び出すだけで構いません。

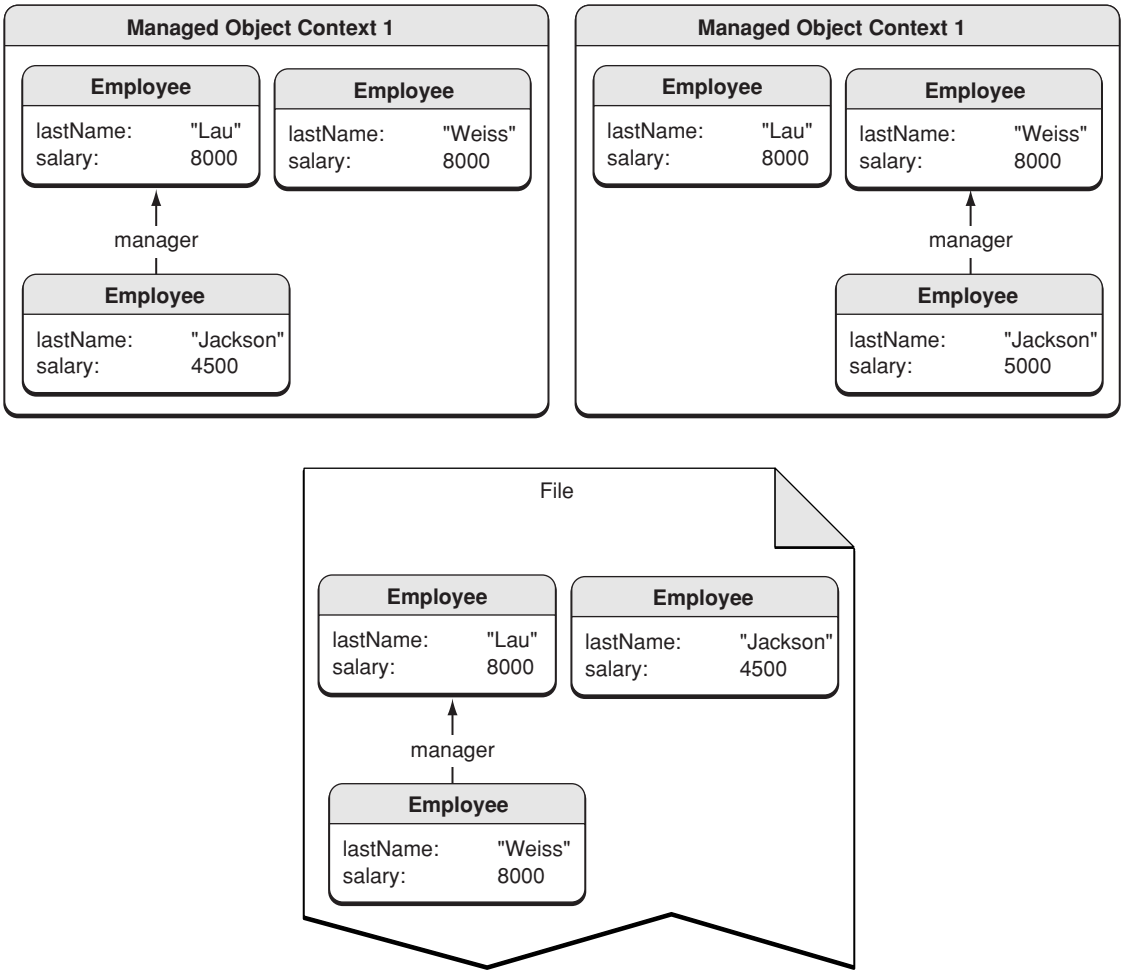
変更管理

アプリケーションに複数の管理オブジェクトコンテキストがあり、どちらでもオブジェクトを修正できるようになっている場合、変更点の違いを調整できるようになっている必要があります。

別々のコンテキストにおける編集

それぞれの管理オブジェクトコンテキスト内では、オブジェクトグラフの整合性が保たれていなければなりません。しかし、同一アプリケーション中に複数の管理オブジェクトコンテキストがある場合、永続ストアの同じレコードを表すオブジェクトがそれぞれのコンテキストに存在し、そのデータが食い違っているということが起こりえます。たとえば、従業員管理アプリケーションの独立した2つのウインドウに、同じ従業員に関するデータが表示されているけれども、別の部署に属している、あるいは上長が違っている、という状況です（図 1を参照）。

図 1　データ値が互いに矛盾している管理オブジェクトコンテキスト



正しい状態は1つしかないので、データを保存する際にビュー間の食い違いを検出し、調整しなければなりません。一方の管理オブジェクトコンテキストを保存すると、永続ストアコーディネータを介して、変更内容が永続ストアに反映されます。その後、もう一方の管理オブジェクトコンテキストを保存しようとすると、**楽観的ロック**と呼ばれる機構で不整合が検出されます。これをどのように解消するかは、コンテキストの設定によります。

不整合の検出と楽観的ロック

Core Dataは、永続ストアからオブジェクトをフェッチすると、その状態の**スナップショット**を作ります。スナップショットの実体は、オブジェクトの永続プロパティの辞書です。通常、属性すべてと、対1関係で結ばれた先のオブジェクトのグローバルIDを、保存するようになっています。このスナップショットが楽観的ロックに関与します。フレームワークは、保存の際、編集した各オブジェクトのスナップショットに記録されている値と、その時点で永続ストアに記録されている値を比較します。

- 値が一致していれば、フェッチした時点から値が変わっていないことになるので、正常に保存できます。その際、スナップショットの値は、永続ストアに一致するよう更新されます。

- 一方、値が違っていれば、オブジェクトをフェッチした、あるいは最後に保存した時点以降に、ストアに変更が施されたことになります。これは楽観的ロックの失敗を意味します。

不整合の解消

複数の永続スタックが同じ外部データストアを参照していれば、楽観的ロックが失敗することがあります（同一アプリケーション内に複数の永続スタックがある場合でも、複数のアプリケーションがある場合でも同じ）。この場合、概念上同じ管理オブジェクトが、同時に2つの永続スタック上で編集された可能性があります。一般的には、今後、第2のスタック上で変更を施すことにより、第1のスタック上で施した変更内容が上書きされて消えてしまう、という事態を回避する方針で実装するところですが、状況によっては、それ以外の方針の方が適切かもしれません。管理オブジェクトコンテキストで適切なマージ方針を選択することにより、どのような動作にするか選択できます。

デフォルトの動作は、`NSErrorMergePolicy`として定義されています。この方針では、マージの際に不整合があると、保存できないようになっています。保存メソッドはこの場合、ユーザ情報（`userInfo`）辞書に「`@"conflictList"`」というキーでエラー情報を格納して戻ります。キーに対応する値は、不整合が生じているレコードの配列です。この配列を調べると、保存しようとしている値と、現時点でストアに格納されている値との差分が分かります。保存するためには、不整合を解消する（オブジェクトを再フェッチしてスナップショットを更新する）か、別の方針を選択しなければなりません。エラーが発生しうる方針は、`NSErrorMergePolicy`だけです。その他の方針、すなわち`NSMergeByPropertyStoreTrumpMergePolicy`、`NSMergeByPropertyObjectTrumpMergePolicy`、`NSOverwriteMergePolicy`の場合、編集したオブジェクトの状態に、ストア内のオブジェクトの状態をそれぞれの方法でマージした上で、保存を続行できます。`NSRollbackMergePolicy`の場合は、メモリ上で行った変更を破棄することにより不整合を解消し、永続ストアにおけるオブジェクトの状態を活かします。

スナップショット管理

数百行に及ぶデータをフェッチするアプリケーションでは、大容量のスナップショットのキャッシュができる場合があります。理論上、Core Dataベースのアプリケーションでは、フェッチ処理を何度も繰り返すと、ストア全体がメモリ上に置かれることになってしまいます。もちろん、このような状況にならないよう、適切にスナップショットを管理しなければなりません。

スナップショットをクリーンアップするのは、**スナップショット参照カウント**という機構の役割です。この機構は、あるスナップショットに関連付けられた管理オブジェクト、すなわち、当該スナップショットから取得したデータが含まれる管理オブジェクトを追跡します。スナップショットに関連付けられた管理オブジェクトのインスタンスがなくなれば（その参照のリストを管理することにより判断）、そのスナップショットを解放するようになっています。

コンテキスト間の変更通知

アプリケーションに複数の管理オブジェクトコンテキストがあっても、あるコンテキストでオブジェクトを変更したとき、自動的にほかのコンテキストに通知されるということはありません。一般にコンテキストとは、ほかから隔離された環境でオブジェクトに変更を施せる「作業領域」を提供するものであり、したがって、必要ならばほかのコンテキストに影響を与えることなく変更内容

を破棄しても構わないからです。コンテキスト間で変更を同期する必要がある場合、変更の取り扱い方は、第2のコンテキストをユーザにどのように見せたいか、オブジェクトの状態はどうか、に依存します。

2つの管理オブジェクトコンテキスト、1つの永続ストアコーディネータから成るアプリケーションを考えてみましょう。第1のコンテキスト（moc1）でユーザがオブジェクトを削除すれば、その旨を第2のコンテキスト（moc2）に通知しなければならないでしょう。どのような場合でも、moc1は NSManagedObjectContextDidSave 通知をポストします。アプリケーションはこの通知を登録し、これをトリガとして必要なアクションを起動しなければなりません。通知には、削除されたオブジェクトだけでなく、変更されたオブジェクトに関する情報も含まれています。削除の結果起こった変更かもしれないので、対処が必要なのです（このような変更がよく起こるものとしては、一時的な関係やフェッチ済みプロパティがあります）。

削除通知の処理法を判断する際には、さまざまな視点からの検討が必要です。特に次の点が重要です。

- 第2のコンテキストに、ほかにどのような変更があるか?
- 削除されたオブジェクトのインスタンスは、第2のコンテキストで変更が施されているか?
- 第2のコンテキストで変更が施されている場合、それは取り消し可能か?

以上はある意味で直交している観点です。そして、コンテキストを同期するために実行するべきアクションは、当該アプリケーションにおけるコンテキストの「意味」によって異なります。以下、3つの戦略を、簡単なものから順に示します。

1. 最も簡単なのは、moc2ではオブジェクト自体が変化しておらず、取り消しについて心配する必要がない場合です。このような状況では、単に削除するだけです。次にmoc2を保存する時点で、フレームワークはオブジェクトを再び削除しようとしている旨を通知し、楽観的ロックの警告を無視して、エラーとはせずに処理を続行します。

2. moc2の内容を気にする必要がないのであれば、単に（resetで）リセットした後、必要なデータを再フェッチして構いません。その結果、取り消しスタックもリセットされ、削除したオブジェクトは復元できなくなります。ここでの問題は、どのデータを再フェッチするべきか、ということだけです。そこで、あらかじめリセット前に、まだ必要な管理オブジェクトのID（objectID）を収集しておき、リセット後、これを使って再読み込みします（このとき、削除済みのIDは除外する必要があります。また、IN述語を組み込んだフェッチ要求を生成することにより、削除したはずのIDにフォールトが起こらないようにするとよいでしょう）。

3. moc2でオブジェクトに変更が施されていても、取り消しについて考慮する必要がないのであれば、アプリケーションにおけるコンテキストの「意味」に応じて戦略を決めて構いません。moc1で削除したオブジェクトが、moc2では変更を施されている場合、moc2でも削除するべきでしょうか?それとも、オブジェクトを復元した上で、変更を保存するべきでしょうか?削除の結果、連鎖的にほかのオブジェクトも削除され、それがmoc2には及んでいない、という場合はどうなるでしょうか?ほかのオブジェクトが削除されたため、連鎖的に当該オブジェクトも削除された、という場合はどうでしょうか?

   これには2つの選択肢があります（実際にはもう1つ、あまりよい戦略とは言えないものがありますが、後述します）。

   a. 最も簡単な戦略は、オブジェクトを削除して、変更した内容を破棄してしまうことです。

b. オブジェクトがほかのオブジェクトと関係で結ばれていなければ、コンテキストのマージ方針を`NSMergePolicyOverwrite`と設定する方法もあります。すると、データベース上で削除された上に、第2のコンテキストにおける変更が上書きされます。

なお、`moc1`で変更したものも含むすべてが、`moc2`における変更により上書きされることに注意してください。

以上は最善の解決法であり、何かを見落としたとしても、オブジェクトグラフの整合性が崩れる可能性はほとんどありません。ほかにも戦略は考えられますが、いずれも不整合やエラーをもたらす可能性があります。いくつか例を示すので、その内容を把握し、避けるようにしてください。*設計の過程で、以下に示す戦略のいずれかに当てはまることに気がついた場合は、先に示したいずれかのパターンになるよう、アプリケーションアーキテクチャを設計し直さなければなりません。*

1. 上記の3（b）のような状況で、しかしオブジェクトが*ほかのオブジェクトと関係で結ばれて*いるとします。それでも何らかの理由で変更を保存したいとすれば、できうる最善の戦略は、`moc2`に読み込まれたグラフの一部を復元することです。もっとも、これが意味のあることか否かは、アプリケーション次第です。やはりマージ方針を`NSMergePolicyOverwrite`としますが、事前に別途何らかのアプリケーション設計が必要でしょう。「削除された」オブジェクトと関係で結ばれたオブジェクトに対し、何らかの処理も必要になります。

後で無意味な状態になってしまわないよう、オブジェクトを実体化した場合に復元する必要が生じうる関係を、自動的に実体化する必要があります。そして、削除通知を受け取ったときには、削除されたオブジェクトと関係で結ばれたオブジェクトがすべて変化した、とコンテキストが判断し、保存できるようにしなければなりません。その結果、アプリケーションのメモリ消費量が増大します。ごく稀にしか起こりえない状況に備えるため、結局は無駄になる可能性が高いデータを保持することになります。しかも、ややもすればデータベースは、`moc1`が生成しようとしている状態とも、`moc2`が想定しているのとも異なる、混乱状態になってしまいます。たとえば関係のつなぎ替えを誤れば、片方向だけの関係になったり、孤児ノードが生じたりするのです。

2. 2番目に悪いのは、 `moc2`にまで影響を及ぼすことができないほかのオブジェクトに対して変更を行った場合で、オブジェクトそのものに施されている変更は破棄する、そして取り消しについても考慮しなければならない、という状況です。変更が失われてしまうため、コンテキストのリセットもできません。オブジェクトを削除すると、その旨が取り消しスタックに入り、取り消しが可能になります。したがってユーザは、取り消しや再保存ができ、あとは上記の3.に挙げた、アプリケーションにおけるコンテキストの「意味」の問題になります。このような状況について設計時に想定していなければ、状況は悪化します。

解決法として本当に意味があるのは、他とは切り離して、アプリケーションコードで、削除の結果変更されるオブジェクトを追跡することです。ユーザの取り消しイベントも追跡して、削除が取り消されても、改めて削除を「再実行」できるようにする必要があります。変更の影響が広い範囲に及んでいれば、非常に複雑で非効率になります。

3. 最悪なのは、ほかのオブジェクトを変更していてこの変更を破棄することはできず、オブジェクトには保存しておきたい変更があり、さらに取り消しについても考慮しなければならない、という状況です。絶対に対処できないとまでは言いませんが、実装には相当の手間が必要であり、非常に複雑かつ脆弱であると言えます。

永続ストアの特徴

Core Dataでは何種類かの永続ストアを扱えます。この章では、それぞれの永続ストアの特徴や長所を述べ、ある型の永続ストアから別の型に移行する手順を解説します。

重要： Mac OS X v10.4では、永続ストアを表す独立したクラスがありません。ストアのインスタンスに対し、`id`として型を指定できるだけです。したがって、もっぱら永続ストアを扱うAPIもありません。以下に解説する技法の多くはMac OS X v10.4にも適用できますが、型として「`NSPersistentStore *`」と記述してある箇所は、「`id`」と書き換える必要があります。

ストアの型と動作

Core Dataでは、XML、アトミック、SQLiteというディスクベースの3種類の永続ストアと、メモリ内ストアを扱えます（Core Dataには、組み込みのアトミックストアとして、`NSBinaryStoreType`で表されるバイナリストア型もあります。さらに、独自のアトミックストア型を定義することも可能です。「カスタムストア型」（126 ページ）を参照）。アプリケーションコードの観点から見ると、通常、個々のストアの実装詳細を気にする必要はありません。管理オブジェクトおよび永続スタックとの間でやり取りすればよいのです。しかし、どの型のストアを採用するか判断する段階では、型による動作の違いを意識しておく必要があります。

iOS： XMLストアは、iOSでは使えません。

	XML	アトミック	SQLite	メモリ内
速度	低速	高速	高速	高速
オブジェクトグラフ	全体を展開	全体を展開	必要部分のみを展開	全体を展開
その他の特徴	外部でパース可能			バックアップ不要

重要： Core DataはSQLiteをストア型として扱いますが、ストアの形式は（ほかのCore Dataネイティブなストアと同様）非公開です。SQLiteデータベースを、SQLiteに付属するネイティブのAPIで生成し、Core Dataで直接利用することはできません（逆に、Core Dataで生成した既存のSQLiteストアを、SQLiteに付属するネイティブのAPIで操作することも不可）。既存のSQLiteデータベースを使うためには、Core Dataのストアにインポートする必要があります（「効率的なデータのインポート」（153 ページ）を参照）。

ストア特有の動作

Core Dataには各種のストアを抽象化して見せる仕組みが備わっているので、通常、開発プロセスを通して同じストアを使う必要はありません。たとえば、プロジェクトのライフサイクルの初期には、XMLストアを使うのが一般的です。人が読める形式なので、想定通りのデータが入っているかどうか、ファイルを開いて確認できるからです。稼働環境に配備し、大容量のデータセットを格納する段階になれば、SQLiteストアを使うのが普通でしょう。高速であり、オブジェクトグラフ全体をメモリ上に展開する必要がないからです。ストアへの書き込みをアトミックにしたいのであれば、バイナリストアを使うことになるでしょう。しかし、それぞれのストア型に特有の特徴や考慮事項がいくつかあります。以下の各節で、それぞれのストア型について説明します。

カスタムストア型

Mac OS X v10.5以降、独自のアトミックストア型を実装できるようになりました。詳しくは『*Atomic Store Programming Topics*』を参照してください。

In Mac OS X v10.4では、Core Dataスタックと透過的にやり取りするような、独自のオブジェクトストアを実装することはできません。しかし、メモリ内ストアを使って、オブジェクトの永続性を自分で管理することは可能です。データを読み込む前に、メモリ内ストアを生成しておきます。データを読み込む際には、適切なモデルクラスのインスタンスを生成して管理オブジェクトコンテキストに追加し、メモリ内ストアに関連付けることになります（`insertObject:`および`assignObject:toPersistentStore:`を参照）。すると管理オブジェクトは、完全にCore Dataスタックに統合され、取り消し管理などの機能を活用できるようになります。ただし、データを保存する処理は開発者の責任です。`NSManagedObjectContextDidSaveNotification`通知を管理オブジェクトコンテキストから受け取るように登録し、通知を受け取り次第、管理オブジェクトを永続ストアに保存することになります。

セキュリティ

Core Dataは、情報源が信頼できるかなど、永続ストアのセキュリティについては何の保証もしません。ファイルが不正に修正されていないかどうか検出することも不可能です。SQLiteストアはセキュリティ保護の面で、XMLストアやバイナリストアよりも多少は優れていますが、これを採用するだけで安全であるとは考えないでください。また、ストアのメタデータについても、セキュリティの考慮が必要です。ストアデータとは別に、メタデータにアーカイブされたデータが改竄される可能性もあるからです。データのセキュリティを確保したい場合は、ディスクイメージの暗号化などの手段を検討してください。

フェッチの述語と整列記述子

フェッチ処理には、ストアの型による違いがいくつかあります。XML、バイナリ、メモリ内ストアでは、述語や整列記述子の評価はObjective-Cで記述されたコードで実行され、`NSString`の比較メソッドなど、Cocoaのあらゆる機能が使われます。一方、SQLストアの場合は、述語や整列記述子をSQLに変換し、データベース側の機能により評価するようになっています。これは主として性能上の理由によるものですが、Cocoa環境で評価するのではないため、Cocoaに依存する整列記述子（や述語）は動作しません。使用可能な整列セレクタとしては、`compare:`、`caseInsensitiveCompare:`、`localizedCompare:`、`localizedCaseInsensitiveCompare:`、`localizedStandardCompare:`があります（`localizedStandardCompare:`はFinderに準じた整列規則を実装したものなので、多くの人が見慣れた並び順になります）。さらに、SQLiteストアの場合、一時プロパティを基準とした整列はできません。

SQLiteストアの場合、述語にはほかにも制約があります。

- どのようなSQLのクエリでも述語に変換できるわけではありません。
- Mac OS X v10.6になるまで、SQLストアは`MATCHES`演算子に対応していませんでした（ストアから返された値をメモリ上に展開した上でしぼり込みに使うことは可能）。
- 述語のキーパスには対多の要素が1つしか使えません。

  たとえば`toOne.toMany.toMany`、`toMany.toOne.toMany`のように組み合わせて、「集合の集合」を評価するようなことはできないのです。したがって、SQLストアに送信する述語では、`ALL`、`ANY`、`IN`のうち1種類の演算子を、1回使うことしかできません。

- CoreDataは「noindex:」に対応していません（関数式についてはNSPredicateの資料を参照）。これは、SQLiteに渡すクエリで、あるインデックスを使わない旨の指定をするものです。これは主として性能上の理由によります。SQLiteではクエリごとに使えるインデックス数に上限があるので、望ましくないインデックスの優先度を落とすために使うのです。

SQLiteストア

SQLiteストアがサポートするファイルシステム

SQLiteストアは、どのようなファイルシステム上のファイルからでも、データの読み込みは可能です。しかし、一般に、バイト単位のロックを実装していないファイルシステムに、直接書き込むことはできません。DOSファイルシステムや一部のNFSファイルシステムでは、バイト単位のロックが正しく実装されていないので、「<dbfile>.lock」によるロックをしています。また、SMBファイルシステムの場合はflockスタイルのロックを使っています。

要約すると、バイト単位のロックが実装されたファイルシステムでは、読み込み/書き出しを適切に並列実行できます。HFS+、AFP、NFSなどがこれに当たります。簡易版のファイルロック機構を組み込んだファイルシステムでもSQLiteストアは使えますが、複数のプロセスによる並列読み込み/書き出しはできません。SMB、DOSなどがこれに当たります。さらに、WebDAVファイルシステム（iDiskを含む）に書き出すことは*できません*。

レコードを削除してもファイル容量が減らない現象

SQLiteストアからレコードを削除しても、ファイル容量が減るとは限りません。多数の項目が削除され、データベースファイルの「ページ」を解放できれば、SQLiteの自動バキューム処理（不要領域の回収）により、データを再配置してページを減らすことができるので、ファイル容量が削減されます。同様に、単独で複数ページを占めている項目（サムネール画像など）を削除しても、ファイル容量は減ることがあります。

SQLiteファイルはページのコレクションとして配置されています。ページ内のデータはB-ツリーで管理されており、単純な固定長レコードではありません。そのため、ストレージ全体の検索が格段に効率的になります。データやインデックスを単一ファイルにどう格納するのが最適か、SQLiteに委ねることができるからです。さらに、これは、データの整合性を維持するための機構（トランザクション処理、ジャーナル）を実現する基盤でもあります。その代償として、削除を行ったとき、ファイル上に使用されない「穴」が生じることがあります。データを削除し、その後別のデータを追加したとき、その「穴」の部分に新しいデータを挿入するか、バキューム処理を行ってデータを圧縮するかは、実行した操作に応じてSQLiteが判断します。

SQLiteストアの保存動作に関する設定

Core DataがSQLiteストアを保存しても、SQLite側では、すぐにストアファイル全体を更新するわけではありません。この部分的に更新した状態で障害が起こると重大な結果を招く恐れがあるので、確実にファイルを書き込むまでは次の処理を行わないようにすることを考えるかもしれません。残念ながらSQLiteストアの場合、ごく少量の変更を書き込むだけでも、たとえばXMLストアに比べてかなり長時間を要する場合があります（たとえば、XMLファイルであれば100分の1秒未満なのに、SQLiteストアでは0.5秒近くかかる、ということが起こりえます。この問題は、XMLストアやバイナリストアでは起こりません。これらはアトミックなので、データが損失しファイルが破損する可能性は非常に低いのです。特に書き込み処理はアトミックで、新しいデータを正常に書き込んでしまうまで、旧ファイルは削除されません）。

Mac OS Xのfsync： Mac OS Xでは、`fsync`コマンドを実行してもバイト列が正常に書き出された旨を保証できないので、SQLiteは`F_FULLFSYNC`要求をカーネルに送信して、確実にドライブのプラッタに書き出されるようにしています。カーネルは強制的にバッファの内容をすべてドライブに書き込み、ドライブはトラックキャッシュの内容を書き込みます。これをしないと、揮発性メモリにデータが長時間置かれたままになるため、システム障害が発生するとデータが破損する恐れがあります。

Core Dataには、SQLiteのディスク同期動作を制御する、独立した2つのプラグマがあり、性能と信頼性のどちらを優先するか制御できるようになっています。

- `synchronous`はディスク同期の頻度を制御します

  ```
  PRAGMA synchronous FULL [2] / NORMAL [1] / OFF [0]
  ```

- `full_fsync` は実行するディスク同期の種類を制御します

  ```
  PRAGMA fullfsync 1 / 0
  ```

  In Mac OS X v10.5では、デフォルト値は`0`となっています。

プラグマについては、http://sqlite.org/pragma.htmlを参照してください。

どちらのプラグマも、オプション辞書のキーNSSQLitePragmasOptionを使って、ストアを開く際に設定できます。NSSQLitePragmasOption辞書には、キーとしてプラグマ名、文字列値としてオブジェクトを、次の例のように記述します。

```
NSPersistentStoreCoordinator *psc = <#Get a persistent store coordinator#>;

NSMutableDictionary *pragmaOptions = [NSMutableDictionary dictionary];
[pragmaOptions setObject:@"NORMAL" forKey:@"synchronous"];
[pragmaOptions setObject:@"1" forKey:@"fullfsync"];

NSDictionary *storeOptions =
    [NSDictionary dictionaryWithObject:pragmaOptions
forKey:NSSQLitePragmasOption];
NSPersistentStore *store;
NSError *error = nil;
store = [psc addPersistentStoreWithType:NSSQLiteStoreType
            configuration: nil
            URL:url
            options:storeOptions
            error:&error];
```

Mac OS X v10.4： Mac OS X v10.4では、full_fsyncがデフォルトになっています。fsyncコマンドを実行してもバイト列が正常に書き込まれた旨を保証できないので、SQLiteはF_FULLFSYNC要求をカーネルに送信します。カーネルは強制的にバッファの内容をすべてドライブに書き込み、ドライブはトラックキャッシュの内容を書き込みます。

Mac OS X v10.4では、SQLiteベースのストア内の、どのデータをディスクに書き出すかを制御するための設定が2つしかありません。データが破損する危険と、ファイルに保存する時間の、どちらを優先するか制御したい場合は、デフォルトキーcom.apple.CoreData.SQLiteDebugSynchronousの値として、次のいずれかを設定してください。

0: ディスク同期を停止

1: 標準動作

2（デフォルト値）:「fctl FULL_FSYNC」コマンドでディスク同期を実行。時間はかかりますが、確実にデータをディスクに書き出せます。

重要： Mac OS X v10.4と10.5ではデフォルトの動作が違います。Mac OS X v10.4では、SQLiteはFULL_FSYNCでディスク同期を実行するのがデフォルトですが、Mac OS X v10.5ではそうではありません。

Core Dataによる並列処理

バックグラウンドのスレッドやキュー上でCore Dataを動かし、同時に別の処理を実行すると有用な状況がいくつかあります。特に、長時間を要する処理をCore Dataが実行している間も、ユーザインターフェイスの応答性を損ないたくない場合に役立ちます。ただし、Core Dataによる並列処理に当たっては、オブジェクトグラフの整合性が崩れないよう、細心の注意が必要です。

注： 並列処理の組み込みには、スレッド、直列操作キュー、ディスパッチキューなどを使います。簡潔にするため、この章ではいずれも「スレッド」と呼ぶことにします。

Core Dataによる並列処理を導入する場合、アプリケーション環境も検討する必要があります。AppKitもUIKitも、大部分はスレッドセーフではありません。特にMac OS XのCocoaバインディングやコントローラは、スレッドセーフではないので、マルチスレッドの実装は非常に複雑なものになる恐れがあります。

スレッド拘束パターンによる並列処理の実現

Core Dataとの並列プログラミングには、*スレッド拘束*というパターンを推奨します。この場合、各スレッドにはそれぞれ、完全にプライベートな管理オブジェクトコンテキストがなければなりません。

このパターンに従って実装する方法は2通りあります。

1. スレッドごとに独立した管理オブジェクトコンテキストを生成し、単一の永続ストアコーディネータを共有する方法。

   通常はこの方法を推奨します。

2. スレッドごとに独立した管理オブジェクトコンテキストと永続ストアコーディネータを生成する方法。

   並列性は高まりますが、代わりに複雑さやメモリ消費量が増えてしまいます。特にコンテキスト間で変更内容をやり取りしなければならない場合、かなり複雑になります。

管理オブジェクトコンテキストを、これを使うスレッド上に生成する必要があるのです。`NSOperation`を使う場合は、`init`メソッドが呼び出し元と同じスレッド上で実行される、ということに注意してください。したがって、キューの`init`メソッドで、キューの管理オブジェクトコンテキストを生成してはなりません。呼び出し元のスレッドに関連付けられてしまうからです。代わりに、直列キューならば`main`、並列キューならば`start`で生成してください。

スレッド拘束パターンに従って実装する場合、管理オブジェクトや管理オブジェクトコンテキストを、スレッド間で受け渡ししないでください。スレッド境界をまたがって、あるコンテキストから別のコンテキストに管理オブジェクトを「渡す」ためには、次のいずれかの方法で実装します。

- そのオブジェクトID（objectID）を渡し、受け取る側の管理オブジェクトコンテキストでは、objectWithID:またはexistingObjectWithID:error:で取得する方法。

  該当する管理オブジェクトは、あらかじめ保存しておく必要があります。新たに挿入したばかりの管理オブジェクトをほかのコンテキストに渡すことはできません。

- 受け取る側のコンテキストでフェッチを実行する方法。

いずれの場合も、受け取った側のコンテキストで、ローカルに管理オブジェクトを生成します。

NSFetchRequestのメソッド群を使えば、スレッドをまたがるデータのやり取りを、より簡単かつ効率的に行えます。たとえばフェッチ要求は、行データを返す（そして行キャッシュを更新する）ことなく、オブジェクトIDのみを返すよう設定することも可能です。オブジェクトIDをバックグラウンドスレッドから別のスレッドに渡すだけであれば、この方法は有用かもしれません。

通常、管理オブジェクトや管理オブジェクトコンテキストで、ロックを使う必要はありません。しかし、永続ストアコーディネータを複数のコンテキストで共有しており、その上で（ストアを追加するなどの）処理を実行したい場合や、多数の処理を仮想的な単一のトランザクションのように1つのコンテキストに集約したい場合は、永続ストアコーディネータをロックする必要があります。

通知機能によりほかのスレッドでの変更を追跡

あるコンテキストで管理オブジェクトに変更を施しても、ほかのコンテキストの該当する管理オブジェクトには、リフレッシュまたは再フォールト化しない限り、影響が及びません。ほかのスレッドの管理オブジェクトに施された変更を追跡しなければならない場合、2通りの方法がありますが、いずれも通知の機能を使います。説明のため、「A」、「B」という2つのスレッドを考えます。Bに変更が施されたとき、Aにも反映されるようにしたい、としましょう。

通常、スレッド*A*の側で、管理オブジェクトコンテキストに対し、保存の際に通知を受け取る旨（NSManagedObjectContextDidSaveNotification）を登録します。通知を受け取ったとき、そのユーザ情報辞書には、スレッドBで挿入、削除、更新された管理オブジェクトの配列が載っています。しかし、この管理オブジェクトは別のスレッドに結び付いているものなので、直接アクセスすることはできません。代わりに、この通知を引数として、mergeChangesFromContextDidSaveNotification:に渡します（スレッドAのコンテキストに送信）。このメソッドを使えば、コンテキストでは、安全に変更をマージできます。

よりきめ細かい制御が必要ならば、管理オブジェクトコンテキストの*変更*通知（NSManagedObjectContextObjectsDidChangeNotification）が使えます。通知のユーザ情報辞書には、やはり、挿入、削除、更新された管理オブジェクトの配列が載っています。しかしこの場合、通知の登録はスレッド*B*の側ですることになります。通知を受け取ったとき、そのユーザ情報辞書に載っている管理オブジェクトは同じスレッドに結び付いているので、オブジェクトIDにアクセスできます。オブジェクトIDをスレッドAに渡します。これは、スレッドA上のオブジェクトに、適切なメッセージを送信することにより行います。スレッドAではこれを受け取り次第、該当する管理オブジェクトを再フェッチできます。

なお、*変更*通知の送信には、NSManagedObjectContextのprocessPendingChangesメソッドを使います。*main*スレッドはアプリケーションのイベントループに結び付いているので、このスレッドが所有するコンテキスト上のユーザイベントごとに、processPendingChangesが自動的に起動されます。バックグラウンドのスレッドでは、このようなことはありません。メソッドがどの時点で起動されるかは、プラットフォームやリリース版によって変わるので、ある特定の動作を前提としたコードにはしないでください。第2のコンテキストがmainスレッドでなければ、適切な時点で、明

示的にprocessPendingChangesを呼び出さなければなりません（バックグラウンドのスレッドに関しては、アクションの区切りごとなど、処理「サイクル」の言い表し方を決めておく必要があるでしょう）。

バックグラウンドでのフェッチ: UI応答性の改善

executeFetchRequest:error:メソッドは、ハードウェアや作業負荷に応じて、適切に動作を調整するようになっています。Core Dataは、必要ならば追加のプライベートメソッドを生成して、フェッチ性能を最適化しようとします。したがって、絶対フェッチ速度を改善するだけの目的で、バックグラウンドスレッドを生成する必要はありません。とはいえ、フェッチ処理をバックグラウンドのスレッドやキューで行うことにより、ユーザインターフェイスの応答性低下を回避できる、という効用はあります。フェッチ処理が複雑であったり、返すデータが大量であったりしても、その結果が届くたびに、ユーザ側に制御を返し、画面に表示できるのです。

スレッド拘束パターンでは、2つの管理オブジェクトコンテキストに、同じ永続ストアコーディネータが関連付けられています。バックグラウンドスレッド上の、一方の管理オブジェクトコンテキストでオブジェクトをフェッチし、その*オブジェクトID*をもう一方のスレッドに渡します。第2のスレッド（通常はアプリケーションの主スレッド、したがって結果を表示可能）では、第2のコンテキストを使って、当該オブジェクトIDのオブジェクトを実体化します（objectWithID:を使ってオブジェクトのインスタンスを生成）。なお、この技法が使えるのはSQLiteストアの場合に限ります。バイナリストアやXMLストアのデータは、開いた時点で即座にメモリ中に読み込まれるからです。

バックグラウンドスレッドでの保存における盲点

非同期のキューやスレッドがあっても、アプリケーションを停止できなくなるわけではありません（特に、NSThreadベースのスレッドはすべて「デタッチ」されます。詳しくはpthreadに関する資料を参照。プロセスは、デタッチされていないスレッドがすべて終了するまでしか動作しません）。したがって、バックグラウンドスレッドで保存操作をすると、その完了前に、強制的に終了してしまう恐れがあります。したがって、保存がすべて完了するまでアプリケーションが終了しないよう、追加のコードを記述しなければなりません。

スレッド拘束パターンを使わない場合

スレッド拘束パターンを使わない、すなわち、管理オブジェクトやコンテキストをスレッド間で受け渡しする場合、ロックについては*慎重すぎる位に*注意し、マルチスレッディングによってもたらされる利点をほとんど捨てる覚悟で設計しなければなりません。さらに次の事項も検討する必要があります。

- 管理オブジェクトを操作、アクセスする際は常に、関連する管理オブジェクトコンテキストを使ってください。

  Core Dataでは、読み込みは「安全」だが変更は「危険」、という状況になることはありません。*あらゆる*操作が「危険」です。どのような操作も、キャッシュの一貫性を損ない、あるいは実体化を引き起こす可能性があるからです。

- 管理オブジェクト自体は、スレッドセーフではありません。

スレッドをまたがって管理オブジェクトを扱う場合、コンテキストをロックする必要があります（NSLockingを参照）。

管理オブジェクトコンテキストや永続ストアコーディネータをスレッド間で共有する場合、メソッドはすべて、スレッドセーフなスコープから起動しなければなりません。ロックに関しては、管理オブジェクトコンテキストや永続ストアコーディネータのNSLockingメソッド群を使い、独自にミューテックスを実装することは避けてください。このメソッド群は、単にミューテックス機能を提供するだけはありません。フレームワークに、アプリケーションの意図、すなわち、スレッド切り替えが可能な、操作の「区切り」を伝えることができるようになっています。

通常、コンテキストやコーディネータのロックには、tryLockまたはlockを使います。この場合フレームワークは、背後で実行される処理もスレッドセーフであることを保証します。たとえば、スレッドごとに1つずつコンテキストを生成するけれども、すべて同じ永続ストアコーディネータを指すという場合、Core Dataはスレッドセーフな方法でコーディネータにアクセスします（NSManagedObjectContextハンドルのlockメソッド、unlockメソッドを再帰的に使用）。

コンテキストをロック（またはtryLockして成功）した場合、そのコンテキストは、unlockを呼び出すまでの間、保持しておく必要があります。マルチスレッド環境で、コンテキストを適切に保持していないと、デッドロックに陥る恐れがあります。

Core Dataの処理性能

Core Dataは全般的に、処理性能に優れたフレームワークと言えるでしょう。Core Dataを使って既存のアプリケーションを実装し直すと、多くの場合、*はるかに*効率がよくなります。しかしフレームワークの使い方を誤ると、効率は損なわれてしまいます。この章では、Core Dataから最大限の性能を引き出す方法について説明します。

はじめに

Core Dataは、機能豊富で洗練されたオブジェクトグラフ管理フレームワークで、大容量のデータももちろん取り扱い可能です。SQLiteストアは拡張性が高く、数十億もの行、多数の表/列から成る、テラバイト単位のデータベースにも対応できます。エンティティそのものに巨大な属性（ただし「大容量データオブジェクト（BLOB）」（140 ページ）を参照）や多数のプロパティがある場合を除き、1万個のオブジェクトであっても、データセットとしてはむしろ小さい方であると言えます。

非常に小規模なアプリケーションでは、Core Dataによるオーバーヘッドが目立つこともありますが（純然たるCocoa文書ベースのアプリケーションを、Core Data文書ベースのアプリケーションと比較）、Core Dataの豊富な機能が使えるという利点の方が大きいでしょう。わずかなオーバーヘッドだけで、ごく簡単なアプリケーションであっても、取り消しと再実行、検証、オブジェクトグラフの管理、さらには永続ストアにオブジェクトを保存する機能も利用できるのです。こういった機能を独自に実装すれば、オーバーヘッドはむしろ増えてしまうでしょう。アプリケーションの複雑さが増すと、Core Dataのオーバーヘッドは相対的に小さくなり、もたらされる利点は逆に大きくなります（たとえば、大規模なアプリケーションに取り消しや再実行の機能を組み込もうとすれば、かなりの作業を要するのが普通です）。

`NSManagedObject`は、データの保存処理を高度に最適化するため、内部ストレージ機構を使います。特に、モデルを調べることにより得られる、データ型に関する情報を利用して最適化します。キー値コーディングやキー値監視に準拠した方法でデータを格納、検索するならば、`NSManagedObject`を使う方が、ほかのストレージ機構よりも高速です。単純な取得/設定であってもこれは変わりません。Cocoaバインディングを活用する現代的なCocoaアプリケーションでは、Cocoaバインディングがキー値コーディングやキー値監視に依存していれば、同等の効率を実現するデータストレージ機構を独自に構築するのは非常に困難でしょう。

しかし、どのような技術でもそうですが、Core Dataも使い方が不適切だとその能力を発揮できません。Core Dataを導入するだけで、メモリ管理など、基本的なCocoaのパターンについて考慮しなくてもよくなるわけではないのです。永続ストアからデータをフェッチする方法についても検討が必要です。想定通りの性能が得られていない場合は、Sharkなどのプロファイリングツールを使って、どこに問題があるか調べてください（「Performance & Debugging」を参照）。

管理オブジェクトのフェッチ

永続ストアとのやり取り（フェッチ）ごとに、ストアにアクセスし、取得したオブジェクトを永続スタックにマージする、というオーバーヘッドが発生します。複数のフェッチ要求はできるだけまとめて、一度に必要なオブジェクトを取得するようにしてください。また、メモリ上に置くオブジェクト数は最小限にしてください。

フェッチ述語

述語の使い方によって、アプリケーションの性能は大きく左右されます。フェッチ要求に複数の述語を組み合わせて指定する場合、最も大きくしぼり込める述語を先頭に置くことにより、性能を改善できます。特にテキストの照合に関係する述語（`contains`、`endsWith`、`like`、`matches`）では、Unicodeを意識した検索に相当の処理時間を要するため、それが顕著です。テキスト比較とそれ以外の比較が混在している場合は、後者の述語を先に置いてください。たとえば「`(salary > 5000000) AND (lastName LIKE 'Quincey')`」の方が、「`(lastName LIKE 'Quincey') AND (salary > 5000000)`」よりもよいことになります。述語の作り方については『*Predicate Programming Guide*』を参照してください。

フェッチ数の制限

フェッチするオブジェクトの数は、次のように、`setFetchLimit:`メソッドで制限できます。

```
NSFetchRequest *request = [[[NSFetchRequest alloc] init] autorelease];
[request setFetchLimit:100];
```

SQLiteストアを使っている場合、フェッチ数を制限することにより、メモリ上に置く管理オブジェクトのワーキングセットを最小限に抑え、アプリケーション全体の性能も改善できます。

大量のオブジェクトを検索する場合、2回に分けてフェッチすることにより、アプリケーションの外見上の応答性を高めることができます。1回目のフェッチでは、比較的少数（たとえば100個）のオブジェクトを検索し、即座に画面に表示してしまいます。こうしておいて、2回目のフェッチでは該当するオブジェクトをすべて（フェッチ数の制限をしないで）検索します。

Mac OS X v10.6になるまで、フェッチを「バッチ処理」する（データベースの用語で言えば、カーソルを設定する）ことはできませんでした。すなわち、「最初の」100個のオブジェクトを取得し、それに続く100個を取得し、さらに100個を取得する、というようなことはできなかったのです。Mac OS X v10.6以降およびiOSでは、`fetchOffset`を使って、結果集合の部分範囲を管理できるようになりました。

しかし一般には、述語を使って、必要なオブジェクトだけを検索するようお勧めします。

フォールティングの動作

フォールトの発動は、（永続ストアに何度もアクセスしなければならない）かなりコストのかかるプロセスなので、できるだけ避けるべきでしょう。フォールトに対する次のメソッドは、実行しても発動することはありません:`isEqual:`、`hash`、`superclass`、`class`、`self`、`zone`、`isProxy`、

`isKindOfClass:`、`isMemberOfClass:`、`conformsToProtocol:`、`respondsToSelector:`、`retain`、`release`、`autorelease`、`retainCount`、`description`、`managedObjectContext`、`entity`、`objectID`、`isInserted`、`isUpdated`、`isDeleted`、`isFault`。

`isEqual`や`hash`は、実行してもフォールトが発動することはないので、管理オブジェクトをコレクションに置いてもフォールトは発動しません。しかし、コレクションオブジェクトに対してキー値コーディングメソッドを起動すると、管理オブジェクトの`valueForKey:`が起動され、フォールトが発動する可能性があります。さらに、`description`のデフォルトの実装ではフォールトが発動することはありませんが、カスタム`description`メソッドを実装し、その中でオブジェクトの永続プロパティにアクセスしていれば、発動する可能性があります。

なお、管理オブジェクトがフォールトであるとしても、当該オブジェクトのデータがメモリ上にないとは限りません（`isFault`の定義を参照）。

SQLiteストアにおけるフォールトの一括実体化と事前フェッチ

フェッチを実行において、Core Dataがフェッチするのは指定されたエンティティのインスタンスだけです。状況によっては（「フォールティングによるオブジェクトグラフの大きさ制限」（105ページ）を参照）、関係で結ばれた先がフォールトで表されていることがあります。Core Dataは、フォールトのデータがアクセスされると、自動的にそれを解決（発動）しようとします。このように、関係先オブジェクトを遅延読み込みする機構は、メモリ消費量の面ばかりでなく、稀にしか使われない（あるいは非常に大容量の）オブジェクトのフェッチ効率を考慮すると、優れた仕組みと言えるでしょう。しかし、多数のオブジェクトについて個々にフェッチ要求を行う結果、オーバーヘッドが非常に大きくなる可能性もあります。たとえば次のようなモデルを考えてみましょう。

これに対して、次のようなコードで、多数のEmployeeをフェッチし、そのDepartmentの名前を検索するとします。

```
NSFetchRequest * employeesFetch = <#A fetch request for Employees#>
// 要求には述語が組み込まれている -- ここに述語がなければ、
// 恐らくすべてのDepartmentをフェッチすることになる。
NSArray *fetchedEmployees = [moc executeFetchRequest:employeesFetch error:&error];
for (Employee *employee in fetchedEmployees)
{
    NSLog(@"%@ -> %@ department", employee.name, employee.department.name);
}
```

この場合、次のような動作になります。

```
Jack -> Sales [fault fires]
Jill -> Marketing [fault fires]
Benjy -> Sales
Gillian -> Sales
Hector -> Engineering [fault fires]
Michelle -> Marketing
```

ここでは永続ストアに4回（1回はEmployeeを取得するする本来のフェッチ、3回は個々のDepartmentを取得するフェッチ）アクセスすることになり、かなりのオーバーヘッドです（1/2の確率で発動）。

これを回避する手法として、**フォールトの一括実体化**と**事前フェッチ**の2つがあります。

フォールトの一括実体化

オブジェクトのコレクションを対象として、一括実体化が可能です。次の例のように、`IN`演算子が入った述語を使ってフェッチ要求を実行すればよいのです（この述語で、`self`は評価の対象オブジェクトを表します。「述語のフォーマット文字列の構文」を参照）。

```
NSArray *array = [NSArray arrayWithObjects:fault1, fault2, ..., nil];
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"self IN %@", array];
```

Mac OS X v10.5以降では、フェッチ要求を生成する際、`NSFetchRequest`の`setReturnsObjectsAsFaults:`メソッドを使って、管理オブジェクトがフォールトの形で返されないよう指定できます。

事前フェッチ

事前フェッチは、実質的にはフォールトの一括実体化の特別な場合で、ほかのフェッチの直後に実行されます。考え方としては、将来必要になるであろうことを予測する、ということです。オブジェクトをいくつかフェッチしたとき、現時点ではフォールトで表されている関係先オブジェクトも、近い将来必要になることが分かっている場合があります。個々のフォールトがばらばらに発動するという非効率性を避けるため、関係先のオブジェクトを事前にフェッチしておこう、というわけです。

Mac OS X v10.5以降では、`NSFetchRequest`の`setRelationshipKeyPathsForPrefetching:`メソッドに関係のキーパスの配列を指定することにより、要求に該当するエンティティを事前フェッチできます。たとえば、あるEmployeeエンティティに、Departmentエンティティに対する関係が設定されているとしましょう。すべての従業員をフェッチし、それぞれについて氏名と所属部署名を印字する、という処理の場合、Departmentの各インスタンスごとにフォールトが発動するのを防ぐため、次のコード例のようにして、department関係を事前フェッチすることができます。

```
NSManagedObjectContext *context = /* コンテキストを取得する */;
NSEntityDescription *employeeEntity = [NSEntityDescription
    entityForName:@"Employee" inManagedObjectContext:context];
NSFetchRequest *request = [[NSFetchRequest alloc] init];
[request setEntity:employeeEntity];
[request setRelationshipKeyPathsForPrefetching:
    [NSArray arrayWithObject:@"department"]];
```

Mac OS X v10.4では、先に検索したオブジェクトと関係で結ばれているエンティティのインスタンスのみ、というフェッチ要求を生成することにより、フェッチの回数を2回（最小限）に抑えることができます。実装方法（あるいは実装するべきか否か）は、関係の基数によって異なります。

- 逆関係が対1であれば、「`@"%K IN %@"`」という形式の述語を使います。ただし、第1引数は逆関係のキー名、第2引数は関係元オブジェクトの配列を表します。
- 逆関係が対多であれば、まず、着目しているフォールトからオブジェクトIDを収集します（ほかの属性にはアクセスしないよう注意）。次に、「`@"SELF IN %@"`」という形式の述語を生成します。ここで引数は、オブジェクトIDの配列を表します。
- 関係が「多対多」の場合、事前フェッチはお勧めしません。

先に挙げた例の場合、department関係の事前フェッチ処理は次のようになります。

```
NSEntityDescription *deptEntity = [NSEntityDescription entityForName:@"Department"
        inManagedObjectContext:moc];
NSArray *deptOIDs = [fetchedEmployees valueForKeyPath:@"department.objectID"];
NSPredicate *deptsPredicate = [NSPredicate predicateWithFormat:
        @"SELF in %@", deptOIDs];
NSFetchRequest *deptFetch = [[[NSFetchRequest alloc] init] autorelease];
[deptFetch setEntity:deptEntity];
[deptFetch setPredicate:deptsPredicate];
// フェッチを実行...
```

データのアクセス法や表現法が分かっていれば、フェッチ述語を工夫して、フェッチするオブジェクト数を抑制できるかもしれません。しかしこのような工夫は、「変化に弱い」ことに注意してください。アプリケーションの仕様が変わり、別のデータが必要になった場合、誤ったオブジェクトを事前フェッチすることになりかねません。

フォールティング、特に`isFault`の戻り値の意味については、「フォールティングと一意化」（105 ページ）を参照してください。

メモリ消費量のオーバーヘッド削減

管理オブジェクトを一時的に使うだけで充分、という状況もあります。ある属性の平均値を求め、その後、個々の属性値は使わない、というような状況です。にもかかわらず、オブジェクトグラフ、したがってメモリ消費量は増えてしまいます。このオーバーヘッドは、不要になった管理オブジェクトを個々に再フォールト化するか、あるいは管理オブジェクトコンテキストをリセットしてオブジェクトグラフ全体をクリアすることにより、削減できます。また、一般的なCocoaプログラミングに用いられるパターンも使えます。

- 個々の管理オブジェクトは、`NSManagedObjectContext`の`refreshObject:mergeChanges:`メソッドで再フォールト化できます。メモリ上にあるプロパティ値をクリアすることにより、メモリ消費のオーバーヘッドを減らす効果があります（これはプロパティを`nil`にすることとは違います。フォールトが発動すれば、再び値が検索されます。「フォールティングと一意化」（105 ページ）を参照）。

- Mac OS X v10.5では、フェッチ要求を生成する際、`includesPropertyValues`を`NO`とすることにより、プロパティ値を表すオブジェクトの生成をやめ、メモリ消費量のオーバーヘッドを削減できます。ただし、今後も実際のプロパティ値は必要ない、あるいはすでに行キャッシュ内に情報があることが確実である場合にのみ、この指定をしてください。そうでないと、永続ストアに何度もアクセスすることになります。

- `NSManagedObjectContext`の`reset`メソッドで、コンテキストに関連付けられた管理オブジェクトをすべて削除し、コンテキストを生成した当初と同じ状態から始めることができます。もちろん、コンテキストに関連付けられた管理オブジェクトはすべて無効になってしまうので、これを参照している箇所があればすべて破棄し、今後必要になるオブジェクトを再フェッチしなければなりません。

- フェッチその他のAPIから返されたオブジェクトは、通常、Cocoaプログラミングガイドラインに従い、自動解放の設定になっています。多数のオブジェクトについて処理を繰り返す場合、独自の自動解放プールを確保、解放して、きめ細かくメモリ管理しなければならないかもしれません。

- Core Dataの取り消し管理機能を使わない場合は、コンテキストの取り消しマネージャとして`nil`を設定することにより、アプリケーションの動作に必要な資源を減らすことができます。これは特に、バックグラウンドのワーカスレッドや、大容量のインポート操作、バッチ処理の場合に有用です。

- 最後に、デフォルトでは、Core Dataは管理オブジェクトを保持していません（まだ保存していない変更がある場合を除く）。メモリ上に多数のオブジェクトがある場合、なぜ保持されたままになっているのか判断し、適切に対処する必要があります。管理オブジェクトは関係を介して相互に保持し合っており、簡単に循環参照（関係をたどると自分に戻ってくる状況）ができてしまいます。オブジェクトを再フォールト化すれば、この循環参照を断ち切ることができます（やはり`NSManagedObjectContext`の`refreshObject:mergeChanges:`メソッドを使用）。

大容量データオブジェクト（BLOB）

アプリケーションでBLOB（Binary Large OBject、画像データ、音声データなど）を扱う場合も、オーバーヘッドを抑える配慮が必要です。なお、「大容量」、「中容量」、「小容量」という言葉にはっきりした定義はなく、アプリケーションの使い方によって変わります。大まかに言って、キロバイト単位のオブジェクトは「中容量」、メガバイト単位であれば「大容量」と考えてもよいでしょう。データベースに10MBのBLOBがあっても、良好な性能を達成できる場合もあります。一方、百万行単位の表が使われている場合、128バイトのオブジェクトであっても「中容量」のCLOB（Character Large OBject）と判断し、表を分割して正規化する必要があるかもしれません。

一般に、どうしても永続ストアにBLOBを格納するのであれば、SQLiteストアを使ってください。XMLストアやバイナリストアの場合、オブジェクトグラフ全体がメモリ上に展開され、ストアへの書き込みはアトミックなので（「永続ストアの特徴」（125 ページ）を参照）、大容量のオブジェクトを効率的に扱うことはできません。SQLiteであれば、巨大なデータベースでも取り扱い可能です。適切に使えば、100GBに及ぶデータベース、1GBに及ぶ行であっても、良好な性能を確保できます（もちろん1GBのデータをメモリ上に読み込む処理は、保管場所がいかに効率的であっても、相応のコストがかかります）。

BLOBは多くの場合、エンティティの属性を表します。たとえば、「photograph」はEmployeeエンティティの属性である、といった具合です。小容量、中容量のBLOB（やCLOB）の場合、データごとに独立したエンティティを生成し、属性の代わりに対1関係を定義してください。たとえば、EmployeeおよびPhotographのエンティティを生成して1対1の関係で結び、Employeeのphotograph属性の代わりとして、EmployeeからPhotographへの関係を使います。このパターンは、オブジェクトのフォールティングをうまく活用できます（「フォールティングと一意化」（105 ページ）を参照）。photographは、実際に必要が生じた（関係がたどられた）時にのみ検索されます。

しかし、BLOBをファイルシステム上の資源として格納し、リンク（URL、パスなど）を管理できるのであれば、その方が優れています。この構成であれば、BLOBは必要に応じて読み込めます。

性能の解析

SQLiteのフェッチ動作の解析

Mac OS X v10.4.3以降、ユーザデフォルト設定「`com.apple.CoreData.SQLDebug`」により、SQLiteに送られる実際のSQLを、`stderr`に出力できるようになりました（ユーザデフォルト名は大文字と小文字が区別されます）。たとえば次の文字列を引数としてアプリケーションに渡したとしましょう。

```
-com.apple.CoreData.SQLDebug 1
```

デバッグレベル番号が大きいほど、詳しい情報が出力されますが、使い勝手は落ちてしまいます。

この出力を見れば、性能上の問題をデバッグする際に役立ちます。特に、（フォールトが個別に発動しているなど）Core Dataが小さなフェッチ処理を何回も実行している場合、その旨を検出できます。出力を見れば、フェッチ要求を使って実行したフェッチと、フォールトを実体化するために自動で実行されるフェッチを区別できます。

Instruments

With Mac OS X v10.5以降、Instrumentsというアプリケーション（デフォルトでは/Developer/Applications/以下にある）で、アプリケーションの動作を解析できるようになりました。Core Dataに特化したInstrumentsプローブがいくつかあります。

- Core Data Fetches

  `executeFetchRequest:error:`が起動されたときに、要求が対象とするエンティティ、返されたオブジェクトの個数、要した時間などの情報とともに記録します。

- Core Data Saves

  `save:`が起動されたときに、要した時間とともに記録します。

- Core Data Faults

  オブジェクトや関係のフォールトが発動した旨を記録します。オブジェクトの発動の場合は該当するオブジェクト、関係の発動の場合は関係元オブジェクトと関係を添えて記録します。また、いずれの場合も、要した時間を記録します。

- Core Data Cache Misses

  フォールト動作のうち、特にファイルシステムのアクティビティに基づくもの（すなわち、データが見つからなかったために発動したもの）をトレースし、データの検索に要した時間とともに記録します。

どのinstrumentについても、各イベントのスタックトレースが記録されているので、その原因を調べることができます。

アプリケーションの解析に際しては、Core Dataに直接関係しない要因についても考慮しなければなりません。たとえば、搭載メモリ容量、オブジェクトの割り当て、その他のAPI（キー値コーディング/キー値監視など）の使用状況（悪用）、といった要因です。

Core Dataのトラブルシューティング

この章では、Core Data上に構築したアプリケーションでよく発生する障害の概要と、解決するための手がかりを示します。

ここで重要なのは、Core Dataが提供する機能が、Cocoaのほかの部品の機能を利用する形で構築されているということです。診断に当たっては、Core Dataに特有の問題と、ほかのフレームワーク、あるいは実装/構築パターンの誤りに起因する問題を、きちんと区別することが大切です。たとえば必要な性能が得られない場合、それ自体はCore Dataの問題ではなく、メモリ管理や資源保護に関して、Cocoaの規範的な技法に反しているのかもしれません。あるいは、画面が適切に更新されない場合（ユーザインターフェイスの障害）、Cocoaバインディングの構成に問題がある可能性もあります。

オブジェクトのライフサイクルに関する問題

マージできない

問題点: `"Could not merge changes"`というエラーメッセージが現れます。

原因: 2つの管理オブジェクトコンテキストが、同じデータを変更しようとしています。これを「楽観的ロックの失敗」とも言います。

対処法: コンテキストのマージ方針を適切に設定するか、ロックの失敗が起こらないようプログラムを書き換えてください。現時点でオブジェクトにコミットされている値は、`committedValuesForKeys:`で確認できます。そして、`refreshObject:mergeChanges:`で、オブジェクトを再フォールト化（次にデータ値がアクセスされた時点で、改めて永続ストアから取得）できます。

管理オブジェクトを別のストアに対応づけている

問題点: 次のような例外が発生します。

```
<NSInvalidArgumentException> [<MyMO 0x3036b0>_assignObject:toPersistentStore:]:
Can't reassign an object to a different store once it has been saved.
```

原因: ストアに対応づけようとしているオブジェクトは、すでに別のストアに対応づけ、保存済みです。

対処法: オブジェクトを別のストアに移動するためには、新たにインスタンスを生成して旧オブジェクトの情報をコピーし、新しいストアに保存した上で、元のインスタンスを削除してください。

フォールトを実体化できない

問題点: `"Core Data could not fulfill a fault"`というエラーメッセージが現れます。

原因: 該当するオブジェクトのデータが永続ストアから削除されています。

対処法: このオブジェクトを削除してください。

この問題は、次のいずれかの状況で発生します。

（1）

- 当初、管理オブジェクトの参照を保持していた。
- 管理オブジェクトコンテキストを介して、この管理オブジェクトを削除した。
- オブジェクトコンテキストに施した変更内容を保存した。

  この時点で、削除されたオブジェクト（の参照）はフォールトになります。参照そのものが削除されないのは、削除するとメモリ管理規則に反するためです。
- 保持したままになっているこの参照をたどって、属性または関係を検索しようとした。

Core Dataはフォールト化した管理オブジェクトを実体化しようとしますが、ストアから削除されてしまっているため、失敗します。すなわち、同じグローバルIDのオブジェクトがストアにはありません。

（2）

- オブジェクトを管理オブジェクトコンテキストから削除した。
- 当該オブジェクトを関係先とする、ほかのオブジェクトからの関係を削除する際に漏れがあった。
- 変更を保存した。

この時点で、ほかのオブジェクトから当該オブジェクトへの関係を発動しようとしても、失敗してしまう場合があります（関係の設定詳細に依存。これによって関係の保存方法が違うため）。

関係の削除規則は、これを設定したオブジェクト「から」の関係（逆関係を含む）にしか適用されません。多数のオブジェクトを（恐らく無意味に）フェッチしてしまう危険を冒すことなく、あるオブジェクト「に」向かう関係を効率的にすべて解消する手段はないのです。

Core Dataのオブジェクトグラフに現れる関係は「有向」であることを忘れないでください。すなわち、関係「元」と関係「先」があるということです。ある方向の関係があるからと言って、逆関係（その逆方向の関係）もあるとは限りません。その意味でも、関係を削除する際、オブジェクトグラフの整合性を適切に維持する必要があります。

実際上、オブジェクトグラフが適切に設計されていれば、オブジェクトを削除したとき、関係を正しく保つために特別な処理を別途行う必要はほとんどありません。オブジェクトグラフの多くに「入り口点」があって、実質的に、ナビゲーション時のルートノートとして機能し、挿入や削除のイベントもフェッチと同じように当該ノードを起点とするのであれば、削除規則を適切に設定することにより、処理の大部分を委ねることができます。同様に、スマートグループやその他の「カジュアルな」関係は一般に、フェッチ済みプロパティを使って適切に実装されているので、削除の

際にオブジェクトグラフの整合性を維持するために、その入り口点で各種の補助コレクションを管理する必要はありません。フェッチ済みの関係には、これを介して見つかるオブジェクトに関する限り、「永続性」の観念がないからです。

管理オブジェクトが無効になった

問題点: 次のような例外が発生します。

```
<NSObjectInaccessibleException> [<MyMO 0x3036b0>_assignObject:toPersistentStore:]:
The NSManagedObject with ID:#### has been invalidated.
```

原因: フォールトが発動したときに使われるはずのストアを削除してしまったか、または管理オブジェクトのコンテキストに`reset`メッセージを送ったためです。

対処法: このオブジェクトを削除してください。ストアを改めて追加すれば、オブジェクトをフェッチできるようになります。

クラスがキー値コーディングの規約に従っていない

問題点: 次のような例外が発生します。

```
<NSUnknownKeyException> [<MyMO 0x3036b0> valueForUndefinedKey:]:
this class is not key value coding-compliant for the key randomKey.
```

原因: 誤ったキーを使ったか、管理オブジェクトを`initWithEntity:inManagedObjectContext:`ではなく`init`で初期化したためです。

対処法: 正しいキーを使ってください（綴り、大文字と小文字の別を確認するほか、キー値コーディングの規約を『*Key-Value Coding Programming Guide*』で確認）。また、`NSManagedObject`専用の初期化メソッドを使ってください（`initWithEntity:insertIntoManagedObjectContext:`を参照）。

エンティティクラスのカスタムメソッドを起動しても応答しない

問題点: `NSManagedObject`のカスタムサブクラスを使うエンティティを定義し、コード中で次のように、このエンティティのインスタンスを生成して、カスタムメソッドを起動しました。

```
NSManagedObject *entityInstance =
    [NSEntityDescription insertNewObjectForEntityForName:@"MyEntity"
            inManagedObjectContext:managedObjectContext];
[entityInstance setAttribute: newValue];
```

すると次のような実行時エラーが発生します。

```
"2005-05-05 15:44:51.233 MyApp[1234] ***
    -[NSManagedObject setNameOfEntity:]: selector not recognized [self = 0x30e340]
```

原因: モデルの定義で、エンティティのカスタムクラス名の綴りが違っているかもしれません。

対処法: モデルにおけるカスタムクラス名と、実装したカスタムクラス名の綴りを一致させてください。

カスタムアクセサメソッドが起動されず、依存キーも更新されない

問題点: あるエンティティに対応するNSManagedObjectのカスタムサブクラスを定義し、カスタムアクセサメソッド（および依存キー）を実装しました。しかし実行時に、アクセサメソッドは呼び出されず、依存キーも更新されません。

原因: モデルで当該エンティティのカスタムクラスを指定していません。

対処法: モデルで、（NSManagedObjectではなく）当該エンティティに対応するカスタムクラスの名前を指定してください。

フェッチ処理に関する問題

SQLiteストアで、うまく整列ができない

問題点: 次のように、NSStringに定義された比較メソッドを用いる、整列記述子を用意しました。

```
NSSortDescriptor *mySortDescriptor = [[NSSortDescriptor alloc]
        initWithKey:@"lastName" ascending:YES
        selector:@selector(localizedCaseInsensitiveCompare:)];
```

次に、フェッチ要求に直接、あるいは配列コントローラの整列記述子のひとつとして、この記述子を使いました。しかし実行時に、次のようなエラーメッセージが現れます。

```
NSRunLoop ignoring exception 'unsupported NSSortDescriptor selector:
        localizedCaseInsensitiveCompare:' that raised during posting of
        delayed perform with target 3e2e42 and selector 'invokeWithTarget:'
```

原因: フェッチ要求がどのように実行されるかは、ストアに依存します（「管理オブジェクトのフェッチ」（61 ページ）を参照）。

対処法: フェッチを直接実行する場合、Cocoaベースの整列演算子は使えないので、メモリ中に配列の形で返されたものを別途整列してください。配列コントローラを使っているのであれば、NSArrayControllerのサブクラスを定義して、整列記述子をデータベースに渡す代わりに、データをフェッチし、それを整列するように実装します。

保存に関する問題

SQLiteストアへの保存に時間がかかりすぎる

問題点: SQLiteストアを使うと、同じデータを保存するのでも、XMLストアに比べて時間がかかります。

原因: これはおそらく、そうあってしかるべき動作です。SQLiteストアが、データがすべてディスクに正しく書き込まれたか確認する、という設定になっています（「SQLiteストアのディスク同期に関する設定」（128 ページ）を参照）。

対処法: まず、保存に要する時間が、ユーザから見て問題になる程度かどうかを判断してください。恐らくこれが目立つのは、文書を頻繁に（たとえば編集を施すごとに）自動保存するようになっている場合だけです。まず、保存時のストアの動作設定を変更してみてください（full_fsyncをオフ）。次に、編集の都度ではなく、一定時間ごと（たとえば15分おき）に保存するよう変更してみます。必要ならば、バイナリストアなど、別の型のストアに変更することも検討してください。

エンティティがnullであるため文書を保存できない

問題点: Core Data文書ベースのアプリケーションで、保存ができません。文書を保存しようとすると、次のような例外が発生します。

```
Exception raised during posting of notification.  Ignored.  exception: Cannot
perform operation since entity with name 'Wxyz' cannot be found
```

原因: このエラーを出しているのは、`NSObjectController`（またはそのサブクラス）のインスタンスです。Entityモードに設定されているけれども、管理オブジェクトモデルのエンティティ記述（Interface Builderで指定されたエンティティ名に対応するもの）にアクセスできない場合に、このエラーが出ます。要するに、コントローラがEntityモードになっているけれども、エンティティ名が正しくないということです。

対処法: Interface Builderでコントローラをそれぞれ選択し、Command-1キーを押してインスペクタを表示してください。各コントローラの正しいエンティティ名が、画面上部の「Entity Name」欄に表示されます。

retainedDataForObjectID:withContextで例外が発生する

問題点: コンテキストにオブジェクトを追加しました。文書を保存しようとすると、次のようなエラーが発生します。

```
[date] My App[2529:4b03] cannot find data for a temporary oid: 0x60797a0
<<x-coredata:///MyClass/t8BB18D3A-0495-4BBE-840F-AF0D92E549FA195>x-coredata:///MyClass/t8BB18D3A-0495-4BBE-840F-AF0D92E549FA195>
```

これは`-[NSSQLCore retainedDataForObjectID:withContext:]`における例外です。バックトレースは次のようになっています。

```
#1    0x9599a6ac in -[NSSQLCore retainedDataForObjectID:withContext:]
#2    0x95990238 in -[NSPersistentStoreCoordinator(_NSInternalMethods)
_conflictsWithRowCacheForObject:andStore:]
#3    0x95990548 in -[NSPersistentStoreCoordinator(_NSInternalMethods)
_checkRequestForStore:originalRequest:andOptimisticLocking:]
#4    0x9594e8f0 in -[NSPersistentStoreCoordinator(_NSInternalMethods)
executeRequest:withContext:]
#5    0x959617ec in -[NSManagedObjectContext save:]
```

`_conflictsWithRowCacheForObject:`の呼び出しでは、保存しようとしているオブジェクトと、データベースから最後にキャッシュした時点のオブジェクトの状態を比較しています。ほかのコード（スレッド、プロセス、ほかの管理オブジェクトコンテキスト）で、知らない間にこのオブジェクトが変更されていないかどうかを確認しているのです。

新たに挿入されたオブジェクトについては、ほかのスコープには存在しえないのでチェックしません。まだデータベースに書き込まれていないからです。

*原因:*新たに挿入したオブジェクトについて、挿入した旨の状態情報を強制的に「失わせた」後に、変更または削除したことが考えられます。一時オブジェクトIDを`objectWithID:`に渡すと、このような状況が起こりえます。あるいは、挿入したオブジェクトをほかの管理オブジェクトコンテキストに渡したかもしれません。

対処法: 根本原因に応じて、対処法はいくつか考えられます。

- 新たに挿入した（まだ保存していない）オブジェクトを、ほかのコンテキストに渡さないでください。コンテキスト間で受け渡ししてよいのは、保存済みのオブジェクトだけです。
- 新たに挿入したオブジェクトに対して、`refreshObject:`を実行しないでください。
- コンテキストに挿入することのないオブジェクトとの間に、関係を結ばないでください。
- `NSManagedObject`のインスタンスについては、専用の初期化メソッドを使ってください。

保存前（スタックトレースのフレーム#6）の時点で、コンテキストの`updatedObjects`や`deletedObjects`に属するオブジェクトは、一時オブジェクト以外（すなわち、そのIDを`isTemporaryID`に渡したとき、戻り値が`NO`となるもの）でなければなりません。

フェッチ処理のデバッグ

Mac OS X v10.4.3以降、ユーザデフォルト設定「`com.apple.CoreData.SQLDebug`」により、SQLiteに送られる実際のSQLを、`stderr`に出力できるようになりました（ユーザデフォルト名は大文字と小文字が区別されます）。たとえば次の文字列を引数としてアプリケーションに渡したとしましょう。

```
-com.apple.CoreData.SQLDebug 1
```

デバッグレベル番号が大きいほど、詳しい情報が出力されますが、使い勝手は落ちてしまいます。

この出力を見れば、性能上の問題をデバッグする際に役立ちます。特に、（フォールトが個別に発動しているなど）Core Dataが小さなフェッチ処理を何回も実行している場合、その旨を検出できます。ファイルI/Oと同様、少量のフェッチ処理を何度も実行すれば、一度にまとめてフェッチするよりもコストが高くなります。対処法については「フォールティングの動作」（136 ページ）を参照してください。

重要： この情報をリバースエンジニアリングに利用して、SQLiteファイルに直接アクセスすることは*できません*。これはもっぱらデバッグのために用いるツールです。

したがって、ログの細かい出力形式は、予告なしに変更される場合があります。たとえばこの出力をパイプ経由でそのまま解析ツールに流し込んだ場合、どの版のOSでも同じように動作するとは限りません。

管理オブジェクトモデル

「+entityForName: could not locate an NSManagedObjectModel」というメッセージが現れる

問題点: メッセージ文面の通りです。エンティティ記述に従って管理オブジェクトモデルを見つけ、そこからエンティティ情報にアクセスすることができません。

原因: 該当するモデルが、アプリケーションリソースに入っていないかもしれません。あるいは、モデルを読み込む前にアクセスしようとしていることも考えられます。コンテキストの参照が`nil`になっている可能性もあります。

対処法: モデルがアプリケーションリソースに入っていること、Xcodeの対応する「プロジェクトの対象」オプションがオンになっていることを確認してください。

起動したクラスメソッドが正常に動作するためには、エンティティ名とコンテキストが必要であり、エンティティはこのコンテキストを介してモデルを取得します。すなわち、次のような流れです。

コンテキスト → コーディネータ → モデル

一般に、Core Dataを扱っていて上記のような問題が起こった場合、次の事項を確認してください。

- 管理オブジェクトコンテキストが`nil`でないこと

  コンテキストの参照をNibファイルで設定する場合、適切な表現方法またはバインディングを正しく設定してください。

- 独自のCore Dataスタックを管理している場合、管理オブジェクトコンテキストにコーディネータが関連付けられていること（コーディネータを確保した後、`setPersistentStoreCoordinator:`で設定）

- 永続ストアコーディネータに正しいモデルがあること

`NSPersistentDocument`を使っているのであれば、管理オブジェクトモデルは、文書の初期化の際に、`mergedModelFromBundles:`メソッドでインスタンス生成されています。

関係資料には、デバッグ方法やデバッグ用フックに関する情報も載っています。たとえば`NSPersistentDocument`に関するAPIリファレンスの「永続オブジェクトの取得と設定」には、文書に埋め込まれているCore Dataオブジェクトの修正、調査に用いるメソッドがいくつか列挙されています。単に実装をオーバーライドし、superを呼び出してその戻り値を調査するだけでも、起こるであろうこと（あるいは起こらないであろうこと）について、より詳しい情報が得られます。

バインディングの統合

バインディングに関する問題の多くは、Core Dataに特有のものではないので、「Cocoaバインディングのトラブルシューティング」を参照してください。この節では、Core Dataとバインディングの統合によって起こる、若干の問題について説明します。

関係の設定ミューテータ（カスタムメソッド）が配列コントローラから呼び出されない

問題点:「対多関係のカスタムアクセサメソッド」に従って関係の設定ミューテータを実装し、`NSArrayController`のインスタンスの`contentSet`バインディングを、関係にバインドしました。しかし、配列コントローラにオブジェクトを追加したり、ここからオブジェクトを削除したりしても、設定ミューテータメソッドが起動されません。

原因: これはバグです。

対処法: `contentSet`バインディングのキーパスに`self`を追加することにより回避できます。たとえば、「[Department Object Controller].`selection.employees`」にバインドしている箇所では、代わりに「[Department Object Controller].`selection.self.employees`」にバインドしてください。

Nibファイルの読み込み後、オブジェクトコントローラの内容にアクセスできない

*問題点:*オブジェクトコントローラ（`NSObjectController`、`NSArrayController`、`NSTreeController`のいずれかのインスタンス）の内容を操作しようとして、Nibファイルを読み込みましたが、コントローラの内容が`nil`です。

原因: コントローラのフェッチは、（Nibファイルの読み込みにより）その管理オブジェクトコンテキストが設定された後で、遅延実行されます。すなわち、フェッチ処理は、`awakeFromNib`および`windowControllerDidLoadNib:`の後で起こります。

対処法: `fetchWithRequest:merge:error:`を呼び出すことにより、遅延実行を待たずに「強制的に」フェッチできます（「Core DataとCocoaバインディング」（115 ページ）を参照）。

配列コントローラで新規オブジェクトを生成できない

問題点: `NSArrayController`で新規オブジェクトを生成できません。たとえば「`add:`」アクションに割り当てられたボタンをクリックしたとき、次のようなエラーになります。

```
2005-05-05 12:00:)).000 MyApp[1234] *** NSRunLoop
ignoring exception 'Failed to create new object' that raised
during posting of delayed perform with target 123456
and selector 'invokeWithTarget:'
```

*原因:*管理オブジェクトモデルで、エンティティに対してカスタムクラスを指定しているのに、そのクラスが実装されていない可能性があります。

対処法: カスタムクラスを実装するか、エンティティが`NSManagedObject`で表現される旨を指定してください。

配列コントローラにバインドされたテーブルビューに、関係の内容が表示されない

*問題点:*配列コントローラにバインドされたテーブルビューに、ある関係の内容を表示したいのですが、何も表示されず、次のようなエラーになります。

```
2005-05-27 14:13:39.077 MyApp[1234] *** NSRunLoop ignoring exception
'Cannot create NSArray from object <_NSFaultingMutableSet: 0x3818f0> ()
of class _NSFaultingMutableSet - consider using contentSet
binding instead of contentArray binding' that raised during posting of
delayed perform with target 385350 and selector 'invokeWithTarget:'
```

原因: コントローラの`contentArray`バインディングを関係にバインドしています。これに対し、関係は（配列でなく）集合で表されます。

対処法: 関係にはコントローラの`contentSet`バインディングをバインドしてください。

新規オブジェクトが、テーブルビューで現在選択されているオブジェクトの関係に追加されない

問題点: テーブルビューがあって、エンティティのインスタンスのコレクションを表示するようになっています。エンティティにはほかのエンティティとの関係があり、そのインスタンスは第2のテーブルビューに表示されます。各テーブルビューは配列コントローラで管理しています。第2のエンティティのインスタンスを追加しましたが、第1のエンティティの、現在選択しているインスタンスの関係に追加されません。

*原因:*2つの配列コントローラが関係で結ばれていません。そのため、第1の配列コントローラに何かが起こっても、その旨が第2の配列コントローラに伝わらないのです。

対処法: 第2の配列コントローラの`contentSet`バインディングを、第1の配列コントローラで選択された関係を指定するキーパスにバインドしてください。たとえば、第1の配列コントローラがDepartmentエンティティ、第2の配列コントローラがEmployeeエンティティを管理している場合、第2の配列コントローラの`contentSet`バインディングは、「`[Department Controller].selection.employees`」となります。

テーブルビューまたはアウトラインビューが、NSArrayControllerまたはNSTreeControllerオブジェクトをバインドしたとき、最新の内容に維持できない

*問題点:*テーブルビューまたはアウトラインビューがあって、エンティティのインスタンスのコレクションを表示するようになっています。エンティティのインスタンスを追加/削除したとき、テーブルビューがそれに同期して変化しません。

*原因:*コントローラの内容（配列）を自分で管理している場合、配列を修正する方法が、キー値監視に準拠していない可能性があります。

コントローラの内容が自動的にフェッチされる場合、コントローラの「Automatically Prepares Content」フラグがオンになっていないかもしれません。

あるいは、コントローラが適切に設定されていない恐れもあります。

対処法: コントローラの内容（コレクション）を自分で管理している場合は、キー値監視に準拠した方法でコレクションを修正しているかどうか確認してください（「Cocoaバインディングのトラブルシューティング」を参照）。

コントローラの内容が自動的にフェッチされるのであれば、Interface Builderの属性インスペクタで、コントローラの「Automatically Prepares Content」フラグをオンにしてください（`automaticallyPreparesContent`も参照）。するとコントローラは、エンティティの管理オブジェクトコンテキストに対する、オブジェクトの追加/削除動作を追跡するようになります。

どちらも当てはまらない場合は、コントローラが正しく設定されているか（エンティティを正しく設定したか、など）確認してください。

効率的なデータのインポート

この章では、データをCore Dataアプリケーションにインポートし、管理オブジェクトの形で永続ストアに保存する、効率的な手順を解説します。さらに、実践するべき基本的なCocoaのパターン、Core Dataに特有のパターンについて検討します。

Cocoaの基本

これは多くの状況で言えることですが、Core Dataでデータファイルをインポートする場合、Cocoaアプリケーション開発の「標準的な規則」を常に意識することが大切です。（ガベージコレクションではなく）管理メモリ環境を利用する場合、特にこれが重要です。データファイルを何らかの方法でパースしながらインポートする場合、恐らく多数の自動解放オブジェクトを生成することになるでしょう。それには大量のメモリが必要で、ページングが多発してしまいます。Core Dataアプリケーションに限った話ではありませんが、ローカル自動解放プールを使うことにより、メモリ上に置くオブジェクトの増加を抑えることができます（たとえば、データ全体にわたってある処理を繰り返すループにおいて、内部的に自動解放プールを使えば、メインループの数回の反復ごとにオブジェクトを解放し、必要に応じて再生成する形で、容易に実装できます）。また、`alloc`および`init`でオブジェクトを生成し、不要になったら`release`で解放するようにすれば、最初から自動解放プールに置く必要はありません。Core Dataとメモリ管理の連携については、「メモリ消費量のオーバーヘッド削減」（139 ページ）を参照してください。

また、無駄に処理を反復することも避けなければなりません。述語内に変数が含まれる場合を例として、無駄な処理を巧妙に回避する方法を紹介しましょう。次のような形の述語の場合、ループで述語を生成するだけでなく、パースも毎回行うことになります。

```
// employeeID（複数）が一致するものを検索
// 検索するemployeeIDを順次anIDに代入しながら繰り返し
// ループの本体内
NSString *predicateString = [NSString stringWithFormat:
        @"employeeID == %@", anID];

NSPredicate *predicate = [NSPredicate predicateWithFormat:predicateString];
```

変数値を埋め込んで整形した文字列から述語を生成するためには、文字列をパースし、述語および式オブジェクトのインスタンスを生成しなければなりません。同じ述語フォームを繰り返し使い、定数値式の値を変えながら処理する場合は、述語を一度だけ生成し、変数置換を行う方が効率的です（「述語の生成」を参照）。その例を以下に示します。

```
// ループの前
NSString *predicateString = [NSString stringWithFormat
        @"employeeID == $EMPLOYEE_ID"];
NSPredicate *predicate = [NSPredicate predicateWithFormat:predicateString];

// ループの本体内
NSDictionary *variables = [NSDictionary dictionaryWithObject:anID
        forKey:@"EMPLOYEE_ID"];
```

```
NSPredicate *localPredicate = [predicate
predicateWithSubstitutionVariables:variables];
```

ピークメモリ消費量の削減

大量のデータをCore Dataアプリケーションにインポートする場合、メモリ消費のピーク量を抑えるため、ある程度の量を一括してインポートするごとにCore Dataスタックを一掃する、という方法があります。この方式に関する課題や技法については、「Core Dataの処理性能」（135 ページ）（特に「メモリ消費量のオーバーヘッド削減」（139 ページ））および「Core Dataによるメモリ管理」（75 ページ）を参照してください。ここでは便宜のため、概要のみを示します。

一括（バッチ方式）インポート

まず、インポート処理用に専用の管理オブジェクトコンテキストを生成し、取り消しマネージャを`nil`としておきます（コンテキストの生成はそれほどコストのかかる処理ではないので、永続ストアコーディネータをキャッシュすれば、特定の作業のために専用のコンテキストを用意しても問題ありません）。

```
NSManagedObjectContext *importContext = [[NSManagedObjectContext alloc] init];
NSPersistentStoreCoordinator *coordinator = <#Get the coordinator#>;
[importContext setPersistentStoreCoordinator:coordinator];
[importContext setUndoManager:nil];
```

（最初からCore Dataスタックがあれば、永続ストアコーディネータはほかの管理オブジェクトコンテキストから取得できます）取り消しマネージャを`nil`とすると、次のような効果があります。

1. （そもそも挿入その他の変更を取り消すような状況はありえないのに、）取り消しに備えてアクションを記録する、という無駄な処理が発生しません。
2. 取り消しマネージャがないので、変更されたオブジェクトに対する強い参照を管理する必要はなく、したがって用済み後にこれを解放することもありません（「変更/取り消し管理」（77 ページ）を参照）。

一括して（バッチ方式で）データをインポートし、対応する管理オブジェクトを生成してください（一度にインポートする最適な量は、各レコードに関連付けられているデータの量と、メモリ消費量をどの程度に抑えたいか、に依存します）。各バッチ処理の先頭で、新たに自動解放プールを生成します。各バッチの末尾では、管理オブジェクトコンテキストを（`save:`で）保存した後、プールを空にします（保存までの間、コンテキストでは、挿入されたオブジェクトに対する変更内容を保持しておく必要があります）。

この手順を実装したコード例を示します。ただし、実際には適切なエラー処理を組み込む必要があります。

```
NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
NSUInteger count = 0, LOOP_LIMIT = 1000;
NSDictionary *newRecord = nil;
NSManagedObject *newMO = nil;

// 辞書を返すメソッド「nextRecord」が、
// 次にファイルからインポートするべきデータ部分を表していると仮定
```

```
while (newRecord = [self nextRecord]) {
    // newRecordをもとに管理オブジェクトを生成

    count++;
    if (count == LOOP_LIMIT) {
        [importContext save:outError];
        [importContext reset];
        [pool drain];

        pool = [[NSAutoreleasePool alloc] init];
        count = 0;
    }
}
// 残りのレコードを保存
if (count != 0) {
    [importContext save:outError];
    [importContext reset];
}
[pool drain];
```

循環参照に関する問題

管理メモリ環境では、事態を複雑にする問題がほかにもあります（ガベージコレクションを用いるアプリケーションには関係ありません）。管理オブジェクトに関係が設定されていると、ほとんどの場合、循環参照が生じ、回収できなくなってしまいます。インポートの際にオブジェクト間の関係を生成する場合、循環参照を解消して、不要になったオブジェクトを解放できるようにしなければなりません。これは、オブジェクトをフォールト化するか、コンテキスト全体をリセットすることにより行います。詳しくは「関係の循環参照の切断」（76 ページ）を参照してください。

「検索し、見つからなければ生成」する処理の効率的な実装

データをインポートする際によく現れる処理として、「検索し、見つからなければ生成」するというパターンがあります。何らかのデータをもとに管理オブジェクトを生成するのですが、まず該当する管理オブジェクトの有無を判断し、ない場合にのみ生成する、というものです。

一連の入力値に該当するオブジェクトを、ストアに保存されている中からそれぞれ検索したい、という状況は少なくありません。単純に考えるならば、それぞれの値について、フェッチ処理により該当する永続オブジェクトなどの有無を判断する、という手順をループで繰り返せばよいことになります。しかしこのパターンは、拡張性の面で劣ります。実際に性能を計測してみれば、フェッチという処理が、（コレクションを対象とする繰り返し処理に比べ、）ループで繰り返すにはコストがかかることが分かるでしょう。実際、本来O(n)時間で解ける問題が、O(n^2)時間の問題になってしまうのです。

可能ならば、1回目のパスで管理オブジェクトをすべて生成してしまい、2回目のパスで関係を整える方が効率的です。たとえば、重複がないことが分かっている（たとえば初期データセットが空であるため）データをインポートする場合、単に各データを表す管理オブジェクトを生成するだけで、検索は必要ないことになります。あるいは、いずれも関係で結ばれていない「フラットな」データをインポートするのであれば、データすべてに対応する管理オブジェクトをまず生成してしまい、単一の長大なIN述語を使って、重複を排除してから保存するとよいでしょう。

一方、「検索し、見つからなければ生成」するというパターンで処理しなければならない場合（たとえば、各種のデータが混じっており、関係情報と属性情報も混在しているものをインポートする場合）でも、既存のオブジェクトの検索方法は、実行するフェッチの回数を減らすことにより最適化できます。その実現方法は、対象とする参照データの量に依存します。インポートするべき新規オブジェクトが100個である場合に、データベースに格納されているのが2000項目程度であれば、既存のオブジェクトをすべてフェッチしてキャッシュしてもそれほどの負担にはなりません（キャッシュに対して何度も検索処理が発生するのであれば、相対的な負担はますます小さくなります）。しかしデータベースに10万項目あれば、すべてキャッシュに置くにはメモリに対する負担が大きすぎます。

IN述語と整列を組み合わせることにより、フェッチ要求を1回だけに抑えられることがあります。たとえば、従業員ID（文字列）のリストが与えられ、そのうち該当する従業員レコードがデータベースにない者について、Employeeレコードを生成したいとしましょう。以下に示すコードを考えてみてください。ここで、Employeeはエンティティで、name属性があるものとします。また、listOfIDsAsStringはIDのリストで、ストアに該当する項目がなければ、オブジェクトを追加するものとします。

まず、文字列を区切って、IDを配列として取り出し、整列します。

```
// 文字列をパースして名前を取り出し、整列する
NSArray *employeeIDs = [[listOfIDsAsString componentsSeparatedByString:@"\n"]
        sortedArrayUsingSelector: @selector(compare:)];
```

次に、IN演算子の右辺に名前文字列の配列を埋め込んだ述語と、結果が名前文字列の配列と同じ順序になるようにするための整列記述子を生成します（INはSQLのIN演算子と同等で、左辺の項が、右辺に指定されたコレクション内に現れていなければならないことを表します）。

```
// IDが合致するEmployeeをすべて取得するフェッチ要求を生成
NSFetchRequest *fetchRequest = [[[NSFetchRequest alloc] init] autorelease];
[fetchRequest setEntity:
        [NSEntityDescription entityForName:@"Employee"
inManagedObjectContext:aMOC]];
[fetchRequest setPredicate: [NSPredicate predicateWithFormat: @"(employeeID IN
 %@)", employeeIDs]];

// 整列した状態で結果が得られるようにする
[fetchRequest setSortDescriptors: [NSArray arrayWithObject:
        [[[NSSortDescriptor alloc] initWithKey: @"employeeID"
                ascending:YES] autorelease]]];
```

最後にフェッチを実行します。

```
NSError *error = nil;
NSArray *employeesMatchingNames = [aMOC
        executeFetchRequest:fetchRequest error:&error];
```

ここまでの処理で、整列済みの配列が2つ得られました。ひとつは従業員IDの配列で、フェッチ要求に埋め込んだものです。もうひとつは実際にフェッチした管理オブジェクトの配列です。整列済みリストを順に走査しながら、以下の手順を繰り返します。

1. 次のIDとEmployeeを取得する。IDがEmployee IDと合致しない場合は、当該IDに対応するEmployeeを生成する。

2. 次のEmployeeを取得する。IDが合致した場合は、次のIDとEmployeeに進む。

フェッチ要求に渡したIDの数にかかわらず、フェッチは1回だけで、あとは結果として得られた配列を走査しながら繰り返しているだけです。

この節に挙げた例の完全なコードを以下に示します。

```
// 文字列をパースして名前を取り出し、整列する
NSArray *employeeIDs = [[listOfIDsAsString componentsSeparatedByString:@"\n"]
        sortedArrayUsingSelector: @selector(compare:)];

// IDが合致するEmployeeをすべて取得するフェッチ要求を生成
NSFetchRequest *fetchRequest = [[[NSFetchRequest alloc] init] autorelease];
[fetchRequest setEntity:
        [NSEntityDescription entityForName:@"Employee"
inManagedObjectContext:aMOC]];
[fetchRequest setPredicate: [NSPredicate predicateWithFormat: @"(employeeID IN
 %@)", employeeIDs]];

// 整列した状態で結果が得られるようにする
[fetchRequest setSortDescriptors: [NSArray arrayWithObject:
        [[[NSSortDescriptor alloc] initWithKey: @"employeeID"
                ascending:YES] autorelease]]];
// フェッチ処理の実行
NSError *error = nil;
NSArray *employeesMatchingNames = [aMOC
        executeFetchRequest:fetchRequest error:&error];
```

Core Dataに関してよく訊ねられる事項

この章では、Core Dataに関してよく訊ねられる事項に答えます。

管理オブジェクトコンテキストは、何をもとに生成されるのですか?

全面的に、アプリケーションに依存します。`NSPersistentDocument`を用いるCocoa文書ベースのアプリケーションでは、通常、永続文書がコンテキストを生成し、`managedObjectContext`のメソッドを介してアクセスできるようにします。

一方、単一ウインドウアプリケーションの場合、標準的なプロジェクト支援機能を使ってプロジェクトを作成したのであれば、アプリケーションデリゲート（AppDelegateクラスのインスタンス）がやはりコンテキストを生成し、`managedObjectContext`のメソッドを介してアクセスできるようにします。しかしこの場合、コンテキスト（およびCore Dataスタックの残り部分）を生成するためのコードが、テンプレートの一部として自動的に生成されます。

なお、`NSController`のサブクラスのインスタンスを直接使って、フェッチを実行することは避けてください（たとえば`NSArrayController`のインスタンスを生成し、フェッチ実行用に使ってはなりません）。コントローラは、モデルオブジェクトとユーザインターフェイスの間のやり取りを管理するためのものです。モデルオブジェクトのレベルで直接フェッチするためには、管理オブジェクトコンテキストを使ってください。

デフォルトデータが入っているストアを初期化するにはどうすればよいですか?

データを生成する方法と、確実に1度だけインポートする方法の、2つの問題があります。

データの生成方法はいくつかあります。

- 永続ストアを別個に生成し、デフォルトデータを格納して、このストアをアプリケーションリソースとして利用できます。使う際には、ストア全体を適当な場所にコピーするか、デフォルトストアから既存のストアにオブジェクトをコピーしなければなりません。
- 小規模なデータセットであれば、そのためのコードを記述して、管理オブジェクトを直接生成することも可能です。
- データをプロパティリスト（あるいは何らかのファイルベースの表現）の形で作成しておき、アプリケーションリソースとして格納できます。これを使う場合、ファイルを開き、内容を走査しながら管理オブジェクトを生成することになります。

この方法は、iOSでは使えません。Mac OS Xの場合も、どうしても必要な場合に限って使ってください。ファイルをパースしてストアを生成する処理は、かなりのオーバーヘッドになります。オフラインでCore Dataストアを作り、アプリケーションで直接使う方が優っています。

デフォルトデータが確実に1度しかインポートされないようにする方法もいくつかあります。

- iOSを使っているか、Mac OS X用に文書ベースではないアプリケーションを開発している場合、アプリケーションの起動時に、アプリケーションストアのファイルが所定の場所に存在するかどうか調べる、という方法が考えられます。存在しなければ、データをインポートする必要があります。iOSベースの例が『*CoreDataBooks*』に載っています。
- `NSPersistentDocument`を使って文書ベースのアプリケーションを開発している場合は、`initWithType:error:`でデフォルトデータを初期化してください。

ストア（したがってファイル）を作成できるのに、データはインポートできない可能性がある場合は、ストアにメタデータフラグを追加してください。メタデータは、`metadataForPersistentStoreWithURL:error:`を使うと、実際にフェッチを実行するよりも効率的に検査できます（デフォルトデータの値をハードコーディングする必要もありません）。

既存のSQLiteデータベースをCore Dataで使うにはどうすればよいですか?

それはできません。 Core DataはSQLiteを永続ストアの1つの型として扱いますが、データベースの形式は非公開です。SQLiteデータベースを、SQLiteに付属するネイティブのAPIで生成し、Core Dataで直接利用することはできません（逆に、Core Dataで生成した既存のSQLiteストアを、SQLiteに付属するネイティブのAPIで操作することも不可）。既存のSQLiteデータベースを使うためには、Core Dataのストアにインポートする必要があります（「効率的なデータのインポート」（153 ページ）を参照）。

エンティティAからエンティティBへの対多関係があるとします。エンティティAのインスタンスが与えられたとき、これと関係で結ばれたエンティティBのインスタンスをフェッチするにはどうすればよいですか?

その必要はありません。きちんと言えば、関係先インスタンスを明示的にフェッチする必要はなく、単にキー値コーディング、またはエンティティAのインスタンスのアクセサメソッドを使えばよいのです。たとえば「widgets」という関係の場合、同名のアクセサメソッドを定義したカスタムクラスを実装していれば、次のように記述してください。

```
NSSet *asWidgets = [instanceA widgets];
```

あるいは次のようなキー値コーディングでも構いません。

```
NSMutableSet *asWidgets = [instanceA mutableSetValueForKey:@"widgets"];
```

オブジェクトを生成したときと同じ順序でフェッチするにはどうすればよいですか?

永続ストアのオブジェクトは順序づけられていません。通常、コントローラ層またはビュー層で、生成日付などの属性を基準として順序づけすることになります。順序を表すデータがある場合は、それを明示的にモデル化する必要があります。

管理オブジェクトをあるコンテキストから別のコンテキストにコピーするにはどうすればよいですか?

まず、厳密な意味では、オブジェクトをコピーするのではないことに注意してください。概念的には、永続ストアに存在する、同じデータを指す参照を追加で生成していることになります。

管理オブジェクトをあるコンテキストから別のコンテキストにコピーするためには、次の例のように、オブジェクトIDを使います。

```
NSManagedObjectID *objectID = [managedObject objectID];
NSManagedObject *copy = [context2 objectWithID:objectID];
```

関係で結ばれたエンティティの属性値に依存して値が決まるキーがあります。属性値が変わったり、関係がつなぎ替えられたりしたとき、即座に追随するためにはどうすればよいですか?

あるプロパティ値がほかのエンティティの属性に依存するといっても、さまざまな状況があります。ある属性値が変化したときには、それに依存するプロパティ値にも変化した旨の印をつけなければなりません。キー値監視通知を依存先プロパティに送る方法は、Mac OS Xの版と、関係の基数（対1か対多か）によって異なります。

Mac OS X v10.5以降、対1関係の場合

対象がMac OS X v10.5以降で、関係先エンティティとの対1関係がある場合、自動的に通知を送るためには、`keyPathsForValuesAffectingValueForKey:`をオーバーライドするか、依存キーを登録するために定義されたパターンに従い、適切な名前のメソッドを実装する必要があります。

`keyPathsForValuesAffectingValueForKey:`は、たとえば次のようにオーバーライドしてください。

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key
{
    NSSet *keyPaths = [super keyPathsForValuesAffectingValueForKey:key];
```

```
    if ([key isEqualToString:@"fullNameAndDepartment"])
    {
        NSSet *affectingKeys = [NSSet setWithObjects:@"lastName", @"firstName",
                                                     @"department.deptName",
nil];
         keyPaths = [keyPaths setByAddingObjectsFromSet:affectingKeys];
    }
    return keyPaths;
}
```

あるいは、次のようにkeyPathsForValuesAffectingFullNameAndDepartmentを実装しても同じ結果が得られます。

```
+ (NSSet *)keyPathsForValuesAffectingFullNameAndDepartment
{
    return [NSSet setWithObjects:@"lastName", @"firstName",
                                 @"department.deptName", nil];
}
```

Mac OS X v10.4、およびMac OS X v10.5で対多関係の場合

対象がMac OS X v10.4であれば、setKeys:triggerChangeNotificationsForDependentKey:にキーパスを渡すことはできないので、上記の方法は使えません。

Mac OS X v10.5であっても、対多関係であればやはり、keyPathsForValuesAffectingValueForKey:にキーパスを渡すことはできません。たとえば、Departmentエンティティに、Employeeとの対多関係（employees）があり、Employeeにはsalary属性があるとします。Departmentにtotal Salaryという属性を定義し、関係で結ばれたEmployeeすべてのsalaryに依存して値が決まるようにしたいとしましょう。たとえばkeyPathsForValuesAffectingTotalSalaryを実装し、キーとしてemployees.salaryを返したとしても、これは*実現できません*。

いずれの場合も、2通りの解決法があります。

1. キー値監視を使って、親（この例ではDepartment）を、子（この例ではEmployee）すべての該当する属性の監視者として登録できます。子オブジェクトを追加/削除したときは、親を監視者として追加/削除しなければなりません（「キー値監視の登録」を参照）。observeValueForKeyPath:ofObject:change:context:メソッドで、次のコード例のように、依存元の値の変化に応じて依存先の値を更新します。

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
change:(NSDictionary *)change context:(void *)context
{
    if (context == totalSalaryContext) {
        [self updateTotalSalary];
    }
    else
    // ほかのキー値を監視、またはsuperを起動...
}
- (void)updateTotalSalary
{
    [self setTotalSalary:[self valueForKeyPath:@"employees.@sum.salary"]];
}
- (void)setTotalSalary:(NSNumber *)newTotalSalary
{
    if (totalSalary != newTotalSalary) {
```

```
        [self willChangeValueForKey:@"totalSalary"];
        [totalSalary release];
        totalSalary = [newTotalSalary retain];
        [self didChangeValueForKey:@"totalSalary"];
    }
}
- (NSNumber *)totalSalary
{
    return totalSalary;
}
```

2. 親を、アプリケーションの通知センターに、管理オブジェクトコンテキストの監視者として登録できます。親はキー値監視と同様の方法で、子が送信する変更通知に応答しなければなりません。

Xcodeの述語ビルダに、フェッチ済みプロパティの述語のプロパティが表示されないのですが、どうしてでしょうか?

Xcodeの述語ビルダで、フェッチ済みプロパティの述語を作成しようとしているのに、プロパティが表示されない場合、恐らくフェッチ済みプロパティの関係先エンティティが未設定です。

Core Dataの効率はどの程度ですか?

Core Dataの開発者チームが、一般的なCore Dataアプリケーションの実行時性能を、Core Dataを使わずに開発した類似アプリケーションと比較しています。全般的に、Core Dataで実装した方が性能面でも優れています。それでもまだ最適化の余地はあり、チームは精力的に改善作業を進めました。Core Dataを使ってできるだけ高い性能を達成する方法については、「Core Dataの処理性能」（135ページ）を参照してください。

Core DataはEOFに似ているように思えます。違いはどこにありますか?

Core DataとEOF（Enterprise Objects Framework、WebObjectsに同梱される）は、同じ技術を基盤としていますが、開発目標が異なります。EOFはJavaベースのフレームワークで、クライアントとしてデータベースサーバに接続する形になっています。一方、Core DataはObjective-Cベースのフレームワークで、デスクトップアプリケーションの開発を支援するよう設計されています。Core Dataには、EOFにはないさまざまな機能がありますし、逆もまたしかりです。

EOFにしかない機能

EOFでは、カスタムSQL、共有編集コンテキスト、入れ子になった編集コンテキストが使えます。Core Dataには、`EOModelGroup`と同等の機能はありません。もっとも、`NSManagedObjectModel`クラスには、既存のモデルをマージするメソッド、マージしたモデルをバンドルから取得するメソッドがあります。

EOFには、関係の事前フェッチや一括実体化の機能がありますが、Mac OS X v10.4まで、Core Dataでは使えませんでした。Mac OS X v10.5では、フェッチ要求を生成する際、`setRelationshipKeyPathsForPrefetching:`で関係のキーパスを指定することにより、関係先エンティティをフェッチしたとき、同時にフェッチされるようにすることができます。

Core Dataにしかない機能

Core Dataには、フェッチ済みプロパティ、ひとつの管理オブジェクトモデルに複数の設定をする機能、ローカルストア、ストア集約（あるエンティティに関するデータを複数のストアに分散）、プロパティ名や検証警告のカスタマイズとローカル化、プロパティの検証に用いる述語、などの機能があります。

同等のクラスの対応

Core DataとEOFには、同等の機能を備えたクラスが数多くあります。

- `NSManagedObject`と`EOGenericRecord`。
- `NSManagedObjectContext`と`EOEditingContext`。
- `NSManagedObjectModel`と`EOModel`。
- `NSPersistentStoreCoordinator`と`EOObjectStoreCoordinator`。
- `NSEntityDescription`、`NSPropertyDescription`、`NSRelationshipDescription`、`NSAttributeDescription`はそれぞれ、`EOEntity`、`EOProperty`、`EORelationship`、`EOAttribute`に対応。

変更管理

変更の伝播については、EOFとCore Dataで重要な違いがあります。Core Dataでは、対応する管理オブジェクトコンテキストどうしが、EOFの編集コンテキストと同じように「同期を保つ」*わけではありません*。2つの管理オブジェクトコンテキストが同じ永続ストアコーディネータに接続されており、どちらにも「同じ」管理オブジェクトがあるとき、一方の管理オブジェクトに修正を施して保存しても、もう一方が再フォールト化される*わけではありません*（すなわち、変更がほかのコンテキストに伝播しません）。ほかの管理オブジェクトを修正し、保存すると、（少なくともデフォルトのマージ方針を採用している場合）楽観的ロックの失敗になります。

マルチスレッド

マルチスレッド環境における、Core Dataの管理オブジェクトコンテキストのロックに関する方針は、EOFの編集コンテキストの場合と異なります。

Mac OS Xデスクトップ

以下の事項は、Mac OS Xのデスクトップにのみ関係するものです。

GUIを介してユーザが入力したデータを検証するにはどうすればよいですか?

Core Dataでは、管理オブジェクトコンテキストに`save:`メッセージが送られたとき、管理オブジェクトをすべて検証するようになっています。Core Dataの文書ベースのアプリケーションでは、ユーザが文書を保存しようとしたときがこれに当たります。GUIを介してデータが入力された時点でデータを検証したい場合は、Interface Builderのバインディングインスペクタで、値バインディングの「即座に検証」オプションをオンにしてください。プログラムでバインディングを設定する場合は、バインディングオプション辞書で、キー`NSValidatesImmediatelyBindingOption`の値を（`NSNumber`オブジェクトの）`YES`とします（「バインディングオプション」を参照）。

カスタム検証メソッドの記述方法については、`NSManagedObject`の、サブクラス定義に関する説明を参照してください。

配列コントローラで管理している詳細テーブルビューからオブジェクトを削除しても、オブジェクトグラフから削除されないのはなぜですか?

配列コントローラが、関係先オブジェクトのコレクションを管理している場合、デフォルトでは、削除メソッドは単に、現在選択されているオブジェクトとの関係を切断するだけです。当該オブジェクト自身がオブジェクトグラフから削除されるようにしたい場合は、`contentSet`バインディングの「Deletes Objects On Remove」オプションをオンにしなければなりません。

文書ベースでないアプリケーションでは、どうやって取り消し/再実行機能を組み込めばよいですか?

Core Dataの文書ベースのアプリケーションでは、標準的な`NSDocument`の取り消しマネージャに代わって、文書の管理オブジェクトコンテキストの取り消しマネージャが使われます。デスクトップMac OS X用の、文書ベースでないアプリケーションの場合、ウインドウのデリゲートが、`windowWillReturnUndoManager:`というデリゲートメソッドを使って、管理オブジェクトコンテキストの取り消しマネージャと同等の機能を提供します。ウインドウのデリゲートに管理オブジェクトコンテキスト用のアクセサメソッドがあれば（Core Data Applicationテンプレートを使う場合のように）、`windowWillReturnUndoManager:`の実装は次のようになります。

```
- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)sender {
    return [[self managedObjectContext] undoManager];
}
```

書類の改訂履歴

この表は「*Core Data プログラミングガイド*」の改訂履歴です。

日付	メモ
2011-08-03	古い文書への参照を削除しました。
2010-11-15	オフラインでストアを再作成する場合の注記をFAQに追加しました。
2009-11-17	スカラ値のアクセサメソッドの実装サンプルを修正しました。フォールティングと一意化の内容を改訂しました。
2009-10-19	編集上の変更を追加しました。
2009-08-25	誤字を訂正しました。
2009-08-20	編集上の細かな変更を行いました。
2009-06-04	NSExpressionDescriptionを使用して特定の値を取り出すための説明を追加しました。
2009-02-26	本書のiOS向けの初版。
2008-11-19	取り消し操作の管理についての説明を拡充しました。
2008-02-08	従来のデータインポートおよびメモリ管理についての説明を拡充しました。
	「関係とフェッチ済みプロパティ」（79 ページ）に多対多の関係に関する説明を追加しました。
2007-12-11	誤字を訂正しました。
2007-10-31	Mac OS X 10.5用に更新しました。
2007-08-30	細かな拡充をいくつか行いました。
2007-08-23	内容の大幅な変更を行い、永続ストアの機能についての情報を追加しました。
2007-07-16	メモリ管理に関する内容を拡充しました。NSManagedObjectサブクラスですべてのアクセサメソッドがmutableSetValueForKey:と一緒に使用されるわけではないことを明記しました。
2007-03-15	スレッドオプションについての説明を拡充しました。関係のアクセサメソッドに関する使用上の制約についての注記を追加しました。
2007-02-08	SQLiteストアでサポートされるファイルシステムを明記しました。

日付	メモ
2007-01-08	フェッチ処理のエンティティ継承に関する動作を明確にしました。「管理オブジェクトモデル」の内容を2つに分割しました。
2006-12-05	「Core Dataを使用したメモリ管理」および「Core DataとCocoaバインディング」のFAQを更新しました。
2006-11-09	フォールティングとKVO通知についての説明を「フォールティングと一意化」に追加しました。
2006-10-03	プロパティへのアクセスと修正、および管理オブジェクトの作成と初期化についての説明を拡充しました。
2006-09-05	管理オブジェクトのコピーについての説明を拡充しました。
2006-07-24	トラブルシューティングとマルチスレッド処理の内容を拡充しました。検証に関する内容を追加しました。
2006-06-28	「永続ストア」で細かな改訂を行いました。細かな誤字を訂正しました。
2006-05-23	コピーアンドペーストのセクションでサンプルコードと詳細情報へのリンクを追加しました。
	「始める前に」を追加しました。
2006-04-04	フェッチ要求のテンプレートについてのセクションを「管理オブジェクトモデル」に追加しました。管理オブジェクトのライフサイクルについての説明を拡充しました。
2006-03-08	「変更管理」および「フォールティングと一意化」の内容を拡充しました。SQLiteのデバッグ用フラグの意味を明確にしました。
2006-02-07	SQLログについての注記を「管理オブジェクトのフェッチ」に追加し、テスト駆動開発についての注記を「バージョン管理」に追加しました。
2006-01-10	スレッドに関する基本的な内容を新たに追加しました。主にNSManagedObjectのAPIリファレンスから引用した「管理オブジェクト」という新しい内容を追加しました。
2005-12-06	「フォールティングと一意化」および「永続ストア」を追加しました。
2005-11-09	従来のファイルのインポートに関する内容を追加しました。
2005-10-04	細かな誤字をいくつか訂正しました。
2005-09-08	管理オブジェクトモデルとバージョン管理について説明した新しい内容を追加しました。
2005-08-11	メモリ管理と管理オブジェクトのフェッチに関する内容を追加しました。
2005-07-07	「管理オブジェクトのアクセサメソッド」の導入部分を簡略化しました。
2005-06-04	細かな誤字をいくつか訂正し、またわかりやすい表現にしました。トラブルシューティングに関する内容を追加しました。

日付	メモ
2005-04-29	管理オブジェクトのアクセサメソッドに関する内容を追加しました。「標準以外の属性」でメソッドリストを修正しました。その他の細かな拡充を行いました。
	関係の操作についての説明を追加し、メモリ管理についての説明を拡充しました。

用語解説

属性(attribute) エンティティの単純なプロパティで、一般に、ほかのエンティティではないもの（たとえばEmployeeオブジェクトの「firstName」）。

Core Dataスタック(core data stack) 管理オブジェクトコンテキストにあるオブジェクトの順序つきコレクション。永続オブジェクトストアコーディネータを介して操作し、永続ストアまたはそのコレクションに渡すために使います。スタックは実質的に、永続ストアコーディネータが定義しています（「永続ストアコーディネータ」を参照）。逆にコーディネータは、スタックごとにただ1つ存在します。永続ストアコーディネータを生成すれば、スタックも新たに生成することになります。

エンティティ(entity) データを保持するオブジェクトを表す抽象的な記述。「モデル/ビュー/コントローラ」設計パターンにおける「モデル」と同等。エンティティの構成要素を「属性」と呼びます。ただし、そのうちほかのモデルの参照であるものは「関係」と呼びます。属性と関係を併せた概念が「プロパティ」です。エンティティと管理オブジェクトの関係は、クラスとインスタンスの関係、あるいはデータベースにおけるテーブルと行の関係に相当します。

フォールト(fault) 外部データストアからまだ読み込んでいないオブジェクトを表す、プレースホルダオブジェクト。対1関係の場合は単一のオブジェクト、対多関係の場合はコレクションを表します。

フォールティング(faulting) 必要になった時点で、外部データストアから「透過的」に読み込むこと。

フェッチ(fetch) データを永続ストアから検索すること。データベースのSELECTに似た処理。その結果をもとに、管理オブジェクトのコレクションを生成し、フェッチ要求の発行元である管理オブジェクトコンテキストに登録するようになっています。

フェッチ要求(fetch request) NSFetchRequestのインスタンス。エンティティ、一連の制約（必要な場合; NSPredicateオブジェクトで表現）、整列記述子（NSSortDescriptorのインスタンス）の配列の3要素を指定します。各要素は、データベースのSELECT文における、テーブル名、WHERE句、ORDER BY句にそれぞれ似ています。これを管理オブジェクトコンテキストに送信して実行します。

フェッチ済みプロパティ(fetched property) エンティティのプロパティのうち、フェッチ要求の形で定義されているもの。弱い片方向の関係を表します。たとえばiTunesの動的プレイリストは、あるオブジェクトに属するプロパティとして表せば、フェッチ済みプロパティになります。曲は特定のプレイリストに「属している」わけではありません。それどころか、曲自体は遠隔サーバ上に置かれている場合もあります。曲が削除されたり、遠隔サーバにアクセスできなくなったりしても、プレイリストがそのまま残っていることがありえます（「スポットライト」のライブクエリとも対比してください）。

挿入(inserting) 管理オブジェクトを管理オブジェクトコンテキストに追加する処理。その結果、オブジェクトはオブジェクトグラフの構成要素となり、永続ストアにコミットされることになります。通常、「挿入」と言えば、管理オブジェクトを最初に生成することだけを指します。その後、永続ストア（「永続ストア」を参照）から検索した管理オブジェクトは、フェッチ（「フェッチ」を参照）された、と言います。管理オブジェクトのライフサイクルを通して1度だけ、最初に管理オブジェクトコンテキスト（「管理オブジェクトコンテキスト」を参照）に挿入された時点で起動される、特殊なメソッド（awakeFromInsert）があります。 管理オブ

ジェクトをオブジェクトグラフの構成要素として組み込むためには、管理オブジェクトコンテキストに挿入しなければなりません。管理オブジェクトコンテキストには、管理オブジェクトに対する変更を監視する役割があります（取り消し機能を提供し、オブジェクトグラフの整合性を維持するため）。監視できるのは、新規オブジェクトが挿入された場合に限ります。

キー値コーディング(key-value coding) オブジェクトのプロパティに、間接的にアクセスするためのメカニズム。

管理オブジェクト(managed object) `NSManagedObject`またはそのサブクラスのインスタンスであるオブジェクト。生成後、管理オブジェクトコンテキストに登録する必要があります。

管理オブジェクトコンテキスト(managed object context) `NSManagedObjectContext`のインスタンスであるオブジェクト。`NSManagedObjectContext`オブジェクトは、アプリケーション内のあるオブジェクト空間（あるいは「作業領域」）を表します。その主な仕事は、管理オブジェクトのコレクションを管理することです。これらのオブジェクトは、関連するModelオブジェクトのグループを形成し、1つ以上の永続ストアに関して整合性のある内部ビューを表します。コンテキストは、管理オブジェクトのライフサイクルで中心的な役割を果たす強力なオブジェクトで、ライフサイクル管理（フォールティングを含む）、妥当性検証、逆関係の操作、取り消し/再実行といった機能があります。

管理オブジェクトモデル(managed object model) `NSManagedObjectModel`のインスタンスであるオブジェクト。`NSManagedObjectModel`オブジェクトは、スキーマ、すなわち、アプリケーションで用いるエンティティ（データモデル）のコレクションを記述します。

オブジェクトグラフ(object graph) 相互に関係するオブジェクトのコレクション。Core Dataでは、管理オブジェクトコンテキストに関連付けられています。また、境界部分で関係がフォールト（「フォールト」を参照）で表されている、不完全なオブジェクトグラフもありえます。

楽観的ロック(optimistic locking) データベースの`UPDATE`文に、更新しようとするオブジェクトに対応するスナップショットの構成要素によって決まる、`WHERE`句を指定するようなもの。

永続ストア(persistent store) オブジェクトを格納するリポジトリ（収納場所）。通常は、XML、バイナリなどの形式のファイル、またはSQLデータベースです。ストアの形式をアプリケーションで意識する必要はありません。Core Dataにはメモリ内ストアもあります。これはプロセスが終了すると消えてしまいます。

永続ストアコーディネータ(persistent store coordinator) `NSPersistentStoreCoordinator`のインスタンスであるオブジェクト。永続ストアと管理オブジェクトモデルの設定を関連付け、管理オブジェクトコンテキストに対して、複数の永続ストアを集約して単一のストアのように見せる働きがあります。

プリミティブアクセサ(primitive accessor) 変数値を直接取得/設定するだけで、アクセス通知/変更通知メソッド（`willAccessValueForKey:`、`didChangeValueForKey:`など）を起動しないアクセサメソッド。通常、永続ストアからフェッチしたときに、オブジェクトの変数を初期化するために使います。これにより、カスタムアクセサメソッドの副作用を避けているのです。

プロパティ(property) エンティティの構成要素。属性または関係。プロパティとエンティティの関係は、インスタンス変数とクラスの関係に相当します。

再フォールト化(refault) オブジェクトをフォールトにすること。変数は、次にアクセスされた時点で、該当する永続ストアから再フェッチされます（ただしキャッシュのメカニズムにも依存）。

関係(relationship) あるエンティティから見た、ほかのエンティティの「単一」インスタンスの参照（対1関係）、または、インスタンスの「コレクション」の参照（対多関係）。たとえばEmployeeオブジェクトの「manager」は対1関係です。

スナップショット(snapshot) 永続ストアからフェッチしたエントリの、フェッチした時点における状態を記録したもの。この情報は、フレームワークの楽観的ロックのメカニズムを実現するために使われます。また、一部の永続ストア

では、変更内容をデータ源に戻す際、最後のフェッチ以降に変更された属性のみを更新するために使われます。

一時プロパティ(transient property) エンティティのプロパティのうち、永続データストアには保存されないもの。ただし、メモリ上で取り消し/再実行ができるようにする目的で、変更の過程は記録されます。

一意化(uniquing) オブジェクトグラフに、永続ストアの同じエントリを表す、複数のオブジェクトができないよう保証すること。Core Dataではこれを実現するために、各管理オブジェクトを永続ストアのエントリに対応付けるマッピングに保持している情報を使います。

検証(validation) プロパティ値が妥当であることを確認する処理。型が正しいこと、値が所定の範囲内であることなどを調べます。Core Dataフレームワークには、値をオブジェクトに適用する前に、それが妥当であることを検証するための基盤機能が組み込まれています。検証には、モデルベースの検証、カスタム検証メソッドによる属性検証、更新/挿入/削除時の属性間検証（整合性の確認）の3種類があります。