

# Objective-Cプログラミングの概念



Developer

# 目次

## CocoaおよびCocoa Touchの、基本的なプログラミングの考え方について 7

At a Glance 7

How to Use This Document 7

Prerequisites 8

関連項目 8

## クラスクラスタ (Class Cluster) 9

クラスクラスタを使わない場合：考え方は単純、しかしインターフェイスは複雑 9

クラスクラスタを導入した場合：考え方は単純、さらにインターフェイスも単純 10

インスタンスの生成 10

公開のスーパークラスが複数あるクラスクラスタ 11

クラスクラスタに属するサブクラスの作成 12

サブクラスを別途定義する方法 13

サブクラスの実装例 14

複合オブジェクトを定義する方法 16

複合オブジェクトの例 17

## クラスファクトリメソッド 20

## デリゲートとデータソース 22

デリゲートの仕組み 22

デリゲートメッセージの形式 24

デリゲートとアプリケーションフレームワーク 26

フレームワーククラスのデリゲートになること 27

delegateプロパティに基づくオブジェクトの検索 27

データソース 28

カスタムクラスにデリゲートを実装する手順 28

## イントロスペクション 30

継承関係の評価 30

メソッドの実装とプロトコルへの準拠 31

オブジェクトの比較 32

## オブジェクトの割り当て 35

## オブジェクトの初期化 36

イニシャライザの形式 36

イニシャライザに関する問題 37

イニシャライザの実装 39

複数のイニシャライザ、指定イニシャライザ 42

## Model-View-Controllerパターン 45

MVCオブジェクトの役割と関係 45

モデルオブジェクトはデータや基本的な振る舞いをカプセル化 45

ビューオブジェクトは情報をユーザ向けに表現 46

コントローラオブジェクトはモデルをビューに結びつける 47

役割の兼務 47

Cocoaの各種のコントローラオブジェクト 48

複合デザインパターンとしてのMVC 49

MVCアプリケーションの設計ガイドライン 52

Cocoa (Mac OS X) におけるModel-View-Controller 54

## オブジェクトモデリング 56

エンティティ 57

アトリビュート 57

リレーション 58

リレーションの基数(cardinality)と所有関係(Ownership) 59

プロパティへのアクセス 60

キー 60

値 60

キーパス 61

## オブジェクトの可変性 63

オブジェクトに可変と不変の2種類がある理由 63

可変オブジェクトに関わるプログラミング 65

可変オブジェクトの生成と変換 65

可変インスタンス変数を保存する、または戻り値として返す場合の取り扱い 66

可変オブジェクトの受け取り 67

コレクション中の可変オブジェクト 69

## アウトレット 70

## Receptionistパターン 72

実践的なReceptionistデザインパターン 72

Receptionistパターンが向いている状況 75

## ターゲット-アクション機構 77

ターゲット 77

アクション 78

AppKitフレームワークにおけるターゲット-アクション 80

    コントロール部品、セル、メニュー項目 80

    ターゲットやアクションの設定 81

    AppKitで定義されているアクション 82

UIKitのターゲット-アクション機構 83

## フレームワークの相互乗り入れについて 85

## 書類の改訂履歴 88

# 図、表、リスト

## クラスクラスタ (Class Cluster) 9

- 図 1-1 数値クラス群の素朴な階層 9
- 図 1-2 複雑になった数値クラス階層 10
- 図 1-3 数値クラス群に「クラスクラスタ」パターンを適用した設計 10
- 図 1-4 クラスタオブジェクトを埋め込んだオブジェクト 16
- 表 1-1 クラスクラスタとその公開スーパークラス 11
- 表 1-2 派生メソッドとその実装例 14

## デリゲートとデータソース 22

- 図 3-1 デリゲートの仕組み 23
- 図 3-2 デリゲートが関与する、より現実的な処理の流れ 24
- リスト 3-1 戻り値があるデリゲートメソッドの例 24
- リスト 3-2 voidを返すデリゲートメソッドの例 25

## イントロスペクション 30

- リスト 4-1 classメソッド、superclassメソッドの使い方 30
- リスト 4-2 isKindOfClass:の使い方 31
- リスト 4-3 respondsToSelector:の使用例 32
- リスト 4-4 conformsToProtocol:の使用例 32
- リスト 4-5 isEqual:の使用例 33
- リスト 4-6 isEqual:のオーバーライド 33

## オブジェクトの初期化 36

- 図 6-1 継承チェーンをさかのぼって初期化 41
- 図 6-2 二次イニシャライザと指定イニシャライザの関係 43

## Model-View-Controllerパターン 45

- 図 7-1 複合パターンとしてのMVC (従来型) 50
- 図 7-2 複合パターンとしてのMVC (Cocoa) 51
- 図 7-3 nibファイルの所有者としてこれを管理する調整型コントローラ 52

## オブジェクトモデリング 56

- 図 8-1 従業員管理アプリケーションのオブジェクト図 57
- 図 8-2 従業員のテーブルビュー 58

- 図 8-3 従業員管理アプリケーションに現れる「リレーション」 59
- 図 8-4 リレーションの基数 59
- 図 8-5 従業員管理アプリケーションのオブジェクトグラフ 61
- 図 8-6 従業員のテーブルビューに部署名が表示されている様子 62

## オブジェクトの可変性 63

- リスト 9-1 可変インスタンス変数の不変なコピーを返すコード例 67
- リスト 9-2 書き換えられる可能性があるオブジェクトのスナップショット作成 68

## Receptionistパターン 72

- 図 11-1 キー値監視による更新通知を主操作キューに「跳ね返らせる」様子 73
- リスト 11-1 Receptionistクラスの宣言 73
- リスト 11-2 Receptionistオブジェクトを生成するクラスファクトリメソッド 74
- リスト 11-3 KVO通知の処理 75
- リスト 11-4 Receptionistオブジェクトの生成 75

## ターゲット-アクション機構 77

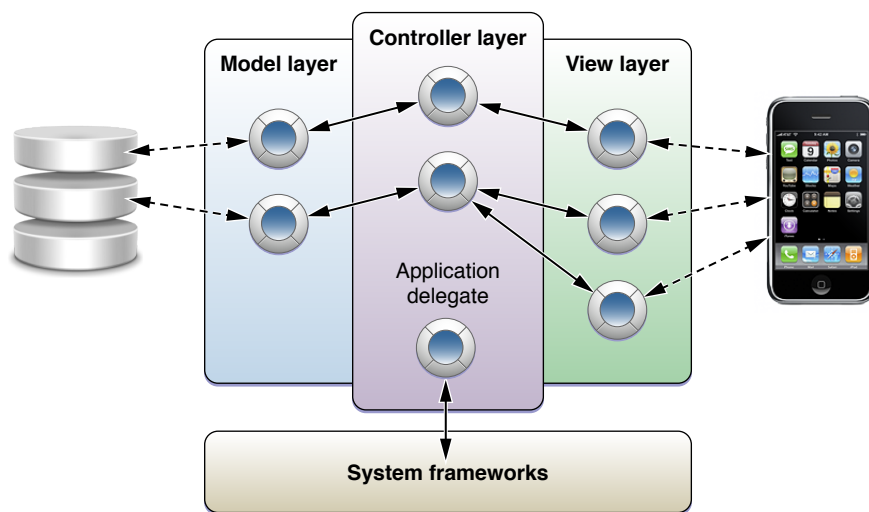
- 図 12-1 コントロール部品とセルから成るアーキテクチャにおける、ターゲット-アクション機構の動作原理 81

## フレームワークの相互乗り入れについて 85

- 表 13-1 Core FoundationとFoundationとで互換性のあるデータ型 86

# CocoaおよびCocoa Touchの、基本的なプログラミングの考え方について

Cocoa / Cocoa Touchフレームワークには多数のプログラミングインターフェイスが組み込まれています。しかしその基盤となる考え方（概念）を知らなければ、どのように扱うべきか理解できないでしょう。この概念はフレームワーク核心部の設計において、基本原理を成すものです。知っていればソフトウェア開発の進め方が違ってくるはずです。



## At a Glance

このドキュメントは、Cocoa / Cocoa Touchフレームワークの中心概念やデザインパターン、機構に関する記事を集めたものです。各記事はアルファベット順に並んでいます。

## How to Use This Document

このドキュメントを通読すれば、Cocoa / Cocoa Touchアプリケーション開発に関する重要事項が身につくはずです。しかし多くの場合、このドキュメントの記事は次のような経緯で目にとまったのでしょ

- ほかのドキュメント（特にiOSやMac OS Xの開発初心者向けドキュメント）からリンクをたどって見つけた。

- 埋め込み小記事（語や字句についている破線の下線をクリックすると現れる記事）に、「最も信頼できる解説」として示されているリンクをたどった。

## Prerequisites

プログラミング（特にオブジェクト指向言語による開発）の経験がある人を対象とします。

## 関連項目

このドキュメントで紹介した概念のうち、言語に関係するものについては、『*The Objective-C Programming Language*』により詳しい解説が載っています。



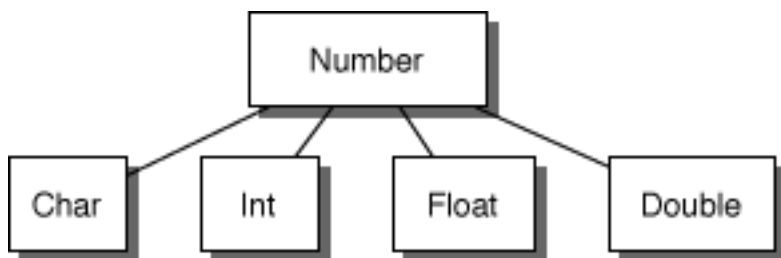
# クラスクラスタ (Class Cluster)

「クラスクラスタ」は、Foundationフレームワークに多用されているデザインパターンです。これは、ある公開抽象スーパークラスから派生した、多数の非公開具象サブクラス群をまとめたもののことです。この方式には、オブジェクト指向フレームワークにおいて、豊富な機能をそのまま提供しつつ、外部に対してより簡潔なアーキテクチャを公開できる、という利点があります。「Abstract Factory」デザインパターンを基盤とした概念です。

## クラスクラスタを使わない場合：考え方は単純、しかしインターフェイスは複雑

クラスクラスタのアーキテクチャとその利点を例示するため、さまざまな型 (char、int、float、double) の数値を格納するクラスの階層を構築する、という問題を取り上げます。それぞれの数値型は、(ある型から別の型に変換できる、文字列として表せるなど) 共通の特性が多いので、すべて同じクラスで表現することもできなくはありません。しかし型によってストレージに対する要件が異なるため、この方式は効率に問題があります。そこで、図 1-1 のようなクラス階層のアーキテクチャにする、という解決方法が思い浮かびます。

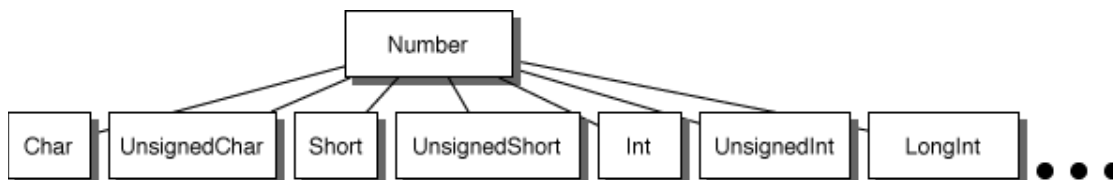
図 1-1 数値クラス群の素朴な階層



Numberは抽象スーパークラスで、各サブクラスに共通のメソッドや演算を宣言しています。しかし、数値を格納するインスタンス変数はありません。サブクラスの側でこの変数を宣言し、Numberに宣言されたプログラムインターフェイスをそれぞれ実装します。

これを見る限りでは、設計は比較的単純です。しかし、Cの基本型にはさまざまな修飾が可能であることを考慮すると、実際のクラス階層は図 1-2 のように複雑になります。

図 1-2 複雑になった数値クラス階層

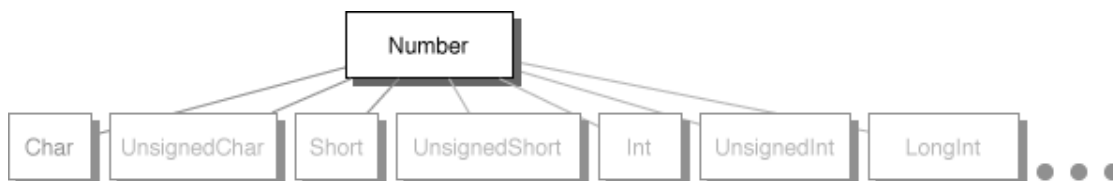


「数値を保持するクラスを個別に定義する」という素朴な考え方では、たちまちクラスの数が増えすぎてしまうのです。これに対し、クラスクラスタの概念を取り入れると、概念の簡潔さをそのまま反映した設計になります。

## クラスクラスタを導入した場合：考え方は単純、さらにインターフェイスも単純

「クラスクラスタ」デザインパターンを適用して設計し直すと、クラス階層は図 1-3 のようになります（非公開クラスは灰色）。

図 1-3 数値クラス群に「クラスクラスタ」パターンを適用した設計



外部から見ると、ここには公開クラスである Number しかありません。個々の型に対応したサブクラスのインスタンスは、どうやって生成するのでしょうか。その答えは、抽象スーパークラスがインスタンス生成を行う、というものです。

## インスタンスの生成

クラスクラスタの抽象スーパークラスには、非公開サブクラスのインスタンス生成メソッドを宣言しなければなりません。生成メソッドが起動されると、スーパークラスは自らの責任で、該当するサブクラスのオブジェクトを生成して返します。呼び出し側でサブクラスを選択するものではありません（したくても不可能です）。

Foundationフレームワークでは一般に、オブジェクトの生成に、「+*className* ...」という形の名前のメソッド、または「alloc...」および「init...」という名前のメソッドを使います。たとえばFoundationフレームワークのNSNumberクラスの場合、次のメッセージを送信して数値オブジェクトを生成します。

```
NSNumber *aChar = [NSNumber numberWithInt:'a'];
NSNumber *anInt = [NSNumber numberWithInt:1];
NSNumber *aFloat = [NSNumber numberWithFloat:1.0];
NSNumber *aDouble = [NSNumber numberWithDouble:1.0];
```

ファクトリメソッドの戻り値として取得したオブジェクトは、明示的に解放する必要がありません。一方、多くのクラスには「alloc...」および「init...」という標準メソッドがあり、これでオブジェクトを生成することも可能ですが、その解放（割り当て解除）は呼び出し側で行う必要があります。

戻り値である各オブジェクト（aChar、anInt、aFloat、aDouble）は、それぞれ異なる非公開サブクラスに属しているかも知れません（実際その通りです）。各オブジェクトが属する具体的なクラスは非公開なので分かりませんが、インターフェイスは抽象スーパークラスNSNumberに宣言があり、公開されています。厳密には正しくないのですが、aChar、anInt、aFloat、aDoubleはすべてNSNumberクラスのインスタンスである、と考えても構いません。生成したのはNSNumberのクラスメソッドですし、アクセスする際にもNSNumberに宣言されたインスタンスメソッドを使うからです。

## 公開のスーパークラスが複数あるクラスクラスタ

先の例では、ひとつの抽象公開クラスに、多数の非公開サブクラスが使うインターフェイスが宣言されていました。これは「純粋な」意味のクラスクラスタと言えます。これに対し、複数の抽象公開クラスに、クラスタで用いるインターフェイスが宣言されている場合があります。むしろその方が望ましいことも多いのです。Foundationフレームワークにはいくつも実例があります（表 1-1を参照）。

表 1-1 クラスクラスタとその公開スーパークラス

クラスクラスタ	公開スーパークラス
NSData	NSData
	NSMutableData
NSArray	NSArray
	NSMutableArray

クラスクラスタ	公開スーパークラス
NSDictionary	NSDictionary
	NSMutableDictionary
NSString	NSString
	NSMutableString

ほかにもこの種のクラスタはありますが、いずれもインターフェイスを2つの抽象クラスに分けて宣言しています。すなわち、どのクラスタオブジェクトも応答できるメソッドを一方の公開クラス、内容の変更を許すオブジェクト（可変オブジェクト）に対してのみ実行できるメソッドをもう一方の公開クラスに宣言しているのです。

このようにクラスタのインターフェイスを分離すると、オブジェクト指向フレームワークにおいて、プログラムの記述能力（表現力）が豊かになります。たとえば書籍を表すオブジェクトに、次のメソッドが宣言されているとしましょう。

```
- (NSString *)title;
```

書籍オブジェクトは、自身のインスタンス変数をそのまま返しても、新たに生成した文字列オブジェクトにコピーして返すように実装しても構いません。戻り値の文字列は変更されない（不変である）ことが、この宣言に明らかに表現されているからです。変更しようとするれば、コンパイラが警告を出します。

## クラスクラスタに属するサブクラスの作成

クラスクラスタのアーキテクチャには、簡明さと拡張性のトレードオフがあります。少数の公開クラスが多数の非公開クラスの代理として振る舞うので、フレームワークを使う側にとっては、使い方を調べるのも実際に使うのも容易になります。一方、クラスタにサブクラスを追加して実装する側にとっては、若干手間が増えてしまうかも知れません。しかし、サブクラスを新たに定義することが滅多にないのであれば、クラスタアーキテクチャは明らかに、利点の多いものになります。Foundationフレームワークでは、まさにこのような状況のときにクラスタを使っています。

必要な機能がクラスタにない場合、サブクラスを別途作成しなければならないかも知れません。たとえば、NSArrayのような配列オブジェクトのクラスクラスタで、ストレージがメモリ上ではなくファイル上にあるものが欲しいとしましょう。この場合、基盤となるストレージ機構を変えることになるので、サブクラスを別に作成しなければなりません。

一方、クラスタから生成したオブジェクトを埋め込むクラスを定義するだけで充分（しかも容易）なこともあります。たとえば、あるデータの値が変わったとき、警告が届くようにしたい、という状況を考えてみましょう。この場合、Foundationフレームワークに定義されたデータオブジェクトをラップする、単純なクラスを用意の方が簡単かも知れません。このラップクラスに、値を変更する旨のメッセージを「横取り」し、（警告を出すなど）独自の処理を実行してから、埋め込みデータオブジェクトに値変更メッセージを転送する、という処理を実装すればよいのです。

以上のように、オブジェクトのストレージを独自の方法で管理する必要があるれば、サブクラスを別途定義してください。そうでなければ、標準的なFoundationフレームワークのオブジェクトを埋め込んだ、複合オブジェクト（ラッパ）を定義するとよいでしょう。以下、この2つの方式について詳しく説明します。

## サブクラスを別途定義する方法

クラスクラスタに、新たにサブクラスを作成する場合、次のように進めてください。

- クラスタの抽象スーパークラスから派生したサブクラスを作成する
- 独自のストレージを宣言する
- スーパークラスのイニシャライザ (initializer) をすべてオーバーライドする
- スーパークラスのプリミティブメソッド（下記参照）をオーバーライドする

クラスタの抽象スーパークラスは、クラスタ階層の中で唯一、外部に公開されているノードなので、第1の条件が必要な理由は明らかです。したがって当然、このサブクラスはクラスタのインターフェイスを継承します。しかしインスタンス変数は、抽象スーパークラスに宣言されていないので、継承することはありません。そこで第2の条件として、サブクラス側に必要なインスタンス変数を宣言しなければならないことになります。最後に、継承したメソッドのうち、インスタンス変数に直接アクセスするものは、すべてオーバーライドしなければなりません。このようなメソッドをプリミティブメソッドと言います。

クラスのプリミティブメソッドは、そのインターフェイスの基盤となります。たとえばNSArrayクラスを考えてみましょう。ここには、オブジェクトの配列を管理するオブジェクトを対象とするインターフェイスが宣言されています。概念的には、配列は数多くのデータ項目を収容し、インデックスを使ってそれぞれの項目にアクセスするようになっています。NSArrayではこの抽象概念を、count および objectAtIndex: という2つのプリミティブメソッドで表しています。この2つを基盤として、ほかのメソッド（派生メソッド）を実装できます。表 1-2 に派生メソッドの例を2つ示します。

表 1-2 派生メソッドとその実装例

派生メソッド	実装例
lastObject	末尾のオブジェクトを取得する。(実装) 配列オブジェクトに[self objectAtIndex: ([self count] - 1)]というメッセージを送信する。
containsObject:	引数と等しいオブジェクトを検索する。(実装) インデックス値を増やしながらobjectAtIndex:メッセージを順次送信し、該当するオブジェクトが見つかるか、すべてのオブジェクトをテストするまで繰り返す。

プリミティブメソッドと派生メソッドを区別することにより、サブクラスの実装が容易になります。継承したプリミティブメソッドさえ適切にオーバーライドすれば、派生メソッドはすべて正常に動作するからです。

プリミティブメソッドと派生メソッドの区別は、初期化が済んだオブジェクトのインターフェイスに関するものです。init...メソッドの取り扱いについては別に検討しなければなりません。

一般に、クラスタの抽象スーパークラスには、多数のinit...メソッド、+ classNameメソッドを宣言することになります。“[インスタンスの生成](#)” (10 ページ) で説明したように、抽象クラス側では、どのinit.../+ classNameメソッドが呼び出されたか、に基づいて、実際に生成するインスタンスの具象サブクラスを判断します。抽象クラスにこういったメソッドをいくつも宣言するのは、サブクラスに対する便宜のためと考えてもよいでしょう。抽象クラスにはインスタンス変数がないので、初期化メソッドは不要です。

サブクラスでは、init...メソッド (インスタンス変数を初期化する必要がある場合) と、場合によっては+ classNameメソッドを、独自に宣言する必要があります。継承するからといって省略することはできません。初期化チェーンのリンクを維持するため、指定イニシャライザ (designated initializer) の実装中に、スーパークラスの対応する指定イニシャライザを起動する記述が必要です。さらに、継承するほかのイニシャライザもすべてオーバーライドし、理に適った動作を実装する必要があります (指定イニシャライザについては『*The Objective-C Programming Language*』の“The Runtime System”を参照)。クラスクラスタでは、抽象スーパークラスの指定イニシャライザは常にinitという名前です。

## サブクラスの実装例

たとえばMonthArrayという名前で、NSArrayのサブクラスを作成することを考えましょう。インデックスに対応する月の名前を返すクラスです。とは言っても、MonthArrayオブジェクトが、月の名前の配列をインスタンス変数として持っているわけではありません。指定されたインデックス位置に相当する名前を返すメソッド (objectAtIndex:) は、定数文字列を返します。したがって、MonthArrayオブジェクトがアプリケーション中にいくつあっても、全体で12個の文字列オブジェクトを割り当てるだけで済みます。

MonthArrayクラスの宣言は次のようになります。

```
#import <foundation/foundation.h>

@interface MonthArray : NSArray

{

}

+ monthArray;
- (unsigned)count;
- (id)objectAtIndex:(unsigned)index;

@end
```

MonthArrayクラスに初期化を要するインスタンス変数はないので、init...メソッドの宣言はないことに注意してください。継承するプリミティブメソッドは、上述のように、count、objectAtIndex:の2つだけです。

MonthArrayクラスの実装は次のようになります。

```
#import "MonthArray.h"

@implementation MonthArray

static MonthArray *sharedMonthArray = nil;
static NSString *months[] = { @"January", @"February", @"March",
    @"April", @"May", @"June", @"July", @"August", @"September",
    @"October", @"November", @"December" };

+ monthArray
{
    if (!sharedMonthArray) {
        sharedMonthArray = [[MonthArray alloc] init];
    }
    return sharedMonthArray;
}
```



```
- (unsigned)count
{
    return 12;
}

- objectAtIndex:(unsigned)index
{
    if (index >= [self count])
        [NSException raise:NSRangeException format:@"%***%s: index
            (%d) beyond bounds (%d)", sel_getName(_cmd), index,
            [self count] - 1];
    else
        return months[index];
}

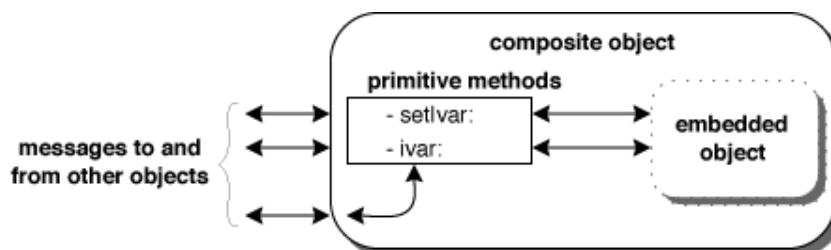
@end
```

MonthArrayでは継承したプリミティブメソッドをオーバーライドしているので、派生メソッドはオーバーライドしなくても正常に動作します。NSArrayには、lastObject、containsObject:、sortedArrayUsingSelector:、objectEnumerator、その他のメソッドがありますが、いずれもMonthArrayオブジェクトでも問題なく動作するのです。

## 複合オブジェクトを定義する方法

独自のオブジェクトを別途用意し、非公開のクラスタオブジェクトをラップする（埋め込む）ことにより、複合オブジェクトを作成できます。この複合オブジェクトは、クラスタオブジェクトの基本機能をそのまま利用し、特別な方法で処理する必要があるメッセージを横取りするだけです。この方式には、コードをそれほど記述することなく、Foundationフレームワークが提供する高品質のコードを活用できる、という利点があります。図 1-4にそのアーキテクチャを示します。

図 1-4 クラスタオブジェクトを埋め込んだオブジェクト





複合オブジェクトは自分自身を、クラスタの抽象スーパークラスから派生したサブクラスとして宣言しなければなりません。したがって、スーパークラスのプリミティブメソッドをオーバーライドする必要があります。派生メソッドもオーバーライドして構いませんが、これはプリミティブメソッドを呼び出す形で動作するので、通常、その必要はありません。

NSArrayクラスのcountメソッドは、オーバーライドすべきプリミティブメソッドの一例です。複合オブジェクト側における実装そのものは、次のように非常に簡単です。

```
- (unsigned)count {  
    return [embeddedObject count];  
}
```

しかし、オーバーライドする側の実装に、独自の処理を組み込むことも可能です。

## 複合オブジェクトの例

複合オブジェクトの例として、可変配列オブジェクトを考えてみましょう。実際に内容を変更する前に、所定の基準でその可否を検査するものとします。次に示すのは、ValidatingArrayというクラスです。標準的な可変配列オブジェクトを埋め込んでいます。ValidatingArrayでは、スーパークラスであるNSArrayおよびNSMutableArrayのプリミティブメソッドを、すべてオーバーライドしています。さらに、インスタンスの生成や初期化に用いる、array、validatingArray、initの各メソッドも宣言しています。

```
#import <foundation/foundation.h>  
  
@interface ValidatingArray : NSMutableArray  
{  
    NSMutableArray *embeddedArray;  
}  
  
+ validatingArray;  
- init;  
- (unsigned)count;  
- objectAtIndex:(unsigned)index;  
- (void)addObject:object;  
- (void)replaceObjectAtIndex:(unsigned)index withObject:object;  
- (void)removeLastObject;  
- (void)insertObject:object atIndex:(unsigned)index;
```

```
- (void)removeObjectAtIndex:(unsigned)index;
```

```
@end
```

実装ファイルを見ると、ValidatingArrayクラスのinitメソッドでは、埋め込みオブジェクトを生成し、変数embeddedArrayに代入していることが分かります。単に配列にアクセスするだけで内容を変更しないメッセージは、単に埋め込みオブジェクトに中継しているだけです。内容を変更するメッセージは、その指定を検査し（ここでは擬似コードのみ）、合格の場合にのみ埋め込みオブジェクトに中継します。

```
#import "ValidatingArray.h"

@implementation ValidatingArray

- init
{
    self = [super init];
    if (self) {
        embeddedArray = [[NSMutableArray allocWithZone:[self zone]] init];
    }
    return self;
}

+ validatingArray
{
    return [[[self alloc] init] autorelease];
}

- (unsigned)count
{
    return [embeddedArray count];
}

- objectAtIndex:(unsigned)index
{

```

```
        return [embeddedArray objectAtIndex:index];
    }

    - (void)addObject:object
    {
        if (/* modification is valid */) {
            [embeddedArray addObject:object];
        }
    }

    - (void)replaceObjectAtIndex:(unsigned)index withObject:object;
    {
        if (/* modification is valid */) {
            [embeddedArray replaceObjectAtIndex:index withObject:object];
        }
    }

    - (void)removeLastObject;
    {
        if (/* modification is valid */) {
            [embeddedArray removeLastObject];
        }
    }

    - (void)insertObject:object atIndex:(unsigned)index;
    {
        if (/* modification is valid */) {
            [embeddedArray insertObject:object atIndex:index];
        }
    }

    - (void)removeObjectAtIndex:(unsigned)index;
    {
        if (/* modification is valid */) {
            [embeddedArray removeObjectAtIndex:index];
        }
    }
}
```

# クラスファクトリメソッド

クラスファクトリメソッドをクラスに実装するのは、クライアントの便宜のためとすることができます。割り当てと初期化を一括して実行し、生成したオブジェクトを返します。しかし、オブジェクトの受け取り手（クライアント）はオブジェクトを所有しないので、（オブジェクト所有の考え方に従い）これを解放する責任も負いません。クラスファクトリメソッドの名前は「+ (type) className ...」という形です（ここで「className」はクラス名から接頭辞を除いたもの）。

Cocoaには、特にデータ値をカプセル化するクラスに、多くの実例があります。たとえばNSDateには、次のようなクラスファクトリメソッドがあります。

```
+ (id)dateWithTimeIntervalSinceNow:(NSTimeInterval)secs;  
+ (id)dateWithTimeIntervalSinceReferenceDate:(NSTimeInterval)secs;  
+ (id)dateWithTimeIntervalSince1970:(NSTimeInterval)secs;
```

また、NSDataには、次のようなファクトリメソッドがあります。

```
+ (id)dataWithBytes:(const void *)bytes length:(unsigned)length;  
+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length;  
+ (id)dataWithBytesNoCopy:(void *)bytes length:(unsigned)length  
    freeWhenDone:(BOOL)b;  
+ (id)dataWithContentsOfFile:(NSString *)path;  
+ (id)dataWithContentsOfURL:(NSURL *)url;  
+ (id)dataWithContentsOfMappedFile:(NSString *)path;
```

ファクトリメソッドは、単なる便宜のためではない目的で実装することもあります。単に割り当てと初期化を一括実行するだけでなく、この2つのプロセス間で情報をやり取りすることができるのです。たとえば、プロパティリストファイルを読み、それに従ってコレクションオブジェクトを初期化しなければならないとしましょう。ファイルには、コレクションの要素（NSStringオブジェクト、NSDataオブジェクト、NSNumberオブジェクト、その他）が多数エンコードされています。ファクトリメソッドでは、コレクション用に必要なメモリ量を見積もるため、ファイルを読んでプロパティリストを解析し（初期化の一部を先取りして実行）、要素の数や、各要素の型を調べます。

あるいはNSWorkspaceクラスのように、インスタンスがシングルトンであることを保証する役割を、クラスファクトリメソッドが担うこともあります。init...メソッドでも、プログラムの実行中を通してインスタンスが1つだけであることを保証できますが、そのためには、「生の」インスタンスをいったん割り当てた後、メモリマネージドコードで解放する必要があります。一方、ファクトリメソッドを実装すれば、使われることのないオブジェクトのために、盲目的にメモリを割り当てる必要はありません（次の例を参照）。

```
static AccountManager *DefaultManager = nil;

+ (AccountManager *)defaultManager {
    if (!DefaultManager) DefaultManager = [[self allocWithZone:NULL] init];
    return DefaultManager;
}
```

# デリゲートとデータソース

デリゲート(delegate, 委譲)とは、あるオブジェクトがプログラム中でイベントに遭遇したとき、それに代わって、または連携して処理するオブジェクトのことです。デリゲートとして指定したオブジェクトが、レスポンドオブジェクトであることも少なくありません。すなわち、AppKitならばNSResponder、UIKitならばUIResponderを継承し、ユーザイベントに応答するオブジェクトです。デリゲートは、当該イベントに関してユーザインターフェイスの制御を委ねられる、あるいは少なくとも、アプリケーション特有の方法でそのイベントを解釈するよう依頼されるオブジェクトです。

デリゲート (delegation) の効用を理解するため、テキストフィールド (NSTextFieldやUITextFieldのインスタンス)、テーブルビュー (NSTableViewやUITableViewのインスタンス) などといった、既製のCocoaオブジェクトと比較してみるとよいでしょう。こういったオブジェクトは、与えられた役割を、汎用的なやり方で果たすよう設計されています。たとえばAppKitフレームワークのウィンドウオブジェクトは、コントロール部品に対するマウス操作に応答し、ウィンドウを閉じる、大きさを変える、移動する、などと解釈して処理します。このように、汎用的でごく限られた振る舞いしかできないので、必然的に、イベントがアプリケーションのほかの部分にどのような影響を及ぼすか、を意識することもあまりありません。特に、アプリケーション特有の振る舞いに対する影響は、まったく意識しないのです。デリゲートは、カスタムオブジェクトがアプリケーション固有の振る舞いを、既製のオブジェクトに伝えるための手段となります。

デリゲートというプログラム機構により、オブジェクトには、プログラムのほかの部分で（通常はユーザアクションに伴って）生じた変化に応じて、自分自身の外観や状態を調整する機会が与えられます。さらに重要なのは、ほかのオブジェクトの振る舞いを、その派生オブジェクトでなくても変更できる、ということです。デリゲートは大部分がカスタムオブジェクトであり、したがって当然、汎用的な（デリゲートの）オブジェクトが認識しえないような、アプリケーション特有のロジックが組み込まれています。

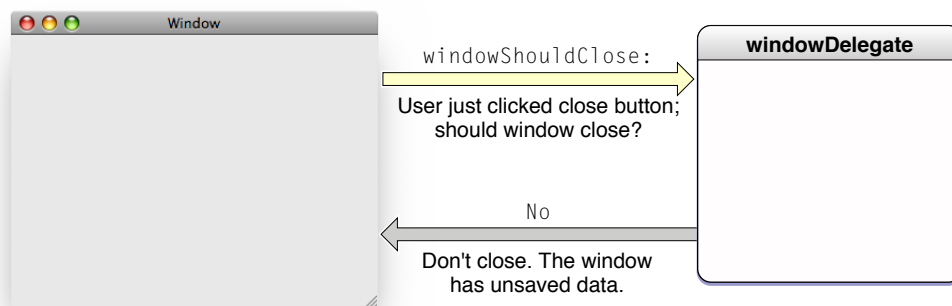
## デリゲートの仕組み

デリゲートの設計は単純です（[図 3-1](#)（23 ページ）を参照）。委譲する側のクラス（デリゲートクラス）には、一般にdelegateという名前の、アウトレットまたはプロパティがあります。アウトレットの場合は、値の設定/取得メソッドもあります。さらに、実装はありませんが、公式 (formal) または非公式 (informal) のプロトコルを構成するメソッドがいくつか宣言されています。省略可のメソッドを用いる公式プロトコル (Objective-C 2.0の機能) の方を推奨しますが、Cocoaフレームワークではどちらのプロトコルも使われています。

非公式プロトコルを使う場合、デリゲートクラス側ではメソッドをNSObjectのカテゴリに宣言し、一方、デリゲート側では、自分自身とデリゲートオブジェクトとが連携するために何らかの形で関与する、あるいはデリゲートオブジェクトのデフォルトの振る舞いに影響を与えるメソッドだけを選んで実装します。一方、デリゲートクラスに公式プロトコルが宣言されている場合、実装が省略可である旨の印がついていれば別ですが、一般に、宣言されたメソッドは実装しなければなりません。

デリゲートの仕組みを利用する場合、図 3-1に示すような設計になります。

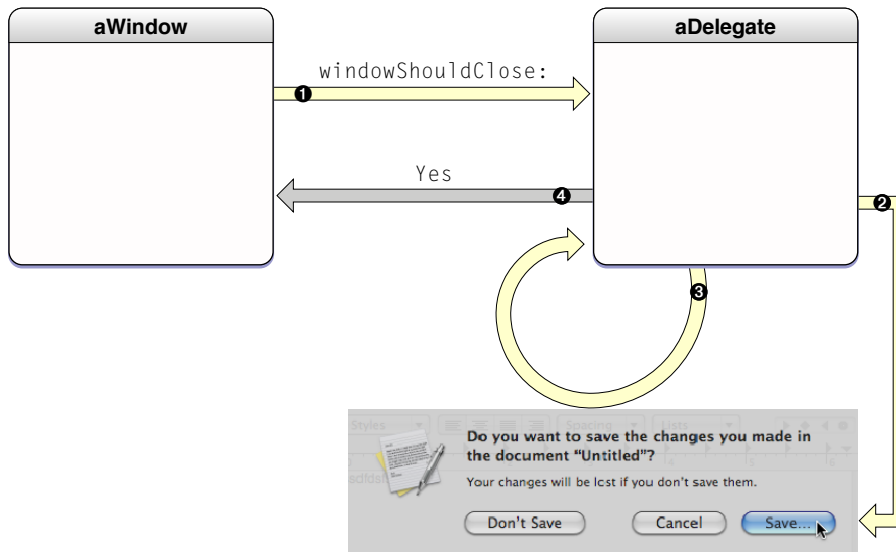
図 3-1 デリゲートの仕組み



プロトコルのメソッドは、デリゲートオブジェクトが処理し、あるいはそう期待される重要なイベントに、その旨の印をつけます。デリゲートオブジェクトは、当該イベントをデリゲート（被委譲オブジェクト）に通知します。あるいは、今まさに発生しようとしているイベントについて、デリゲート側でユーザの入力や承認を求めるよう要求します。たとえば、Mac OS X上のウインドウの「閉じる」ボタンをユーザがクリックすると、ウインドウオブジェクトはwindowShouldClose:メッセージをデ

リゲートに送ります。デリゲートはこの機会を利用して、たとえば未保存のデータがウインドウに関連づけられている場合に、「閉じる」処理を禁止し、あるいは遅延させることができます（図 3-2を参照）。

図 3-2 デリゲートが関与する、より現実的な処理の流れ



デリゲートオブジェクトは、デリゲートに該当するメソッドが実装されている場合に限り、メッセージを送信します。この判断は、NSObjectから継承した、デリゲートのrespondToSelector:メソッドを呼び出すことにより行います。

## デリゲートメッセージの形式

デリゲートメソッドの名前のつけ方には規約があります。メソッド名の先頭は、デリゲートする側の（AppKitまたはUIKitの）オブジェクト、すなわち、アプリケーション、ウインドウ、コントロール部品などの名前です。すべて小文字にし、接頭辞（「NS」、「UI」）を省いて使います。（常にではありませんが）通常はこの後に、報告されたイベントの時制を表す助動詞が続きます。言い替えると、イベントが発生しようとしている（「Should」または「Will」）か、すでに発生したところである（「Did」または「Has」）かを表す助動詞です。このように時制を示すことにより、戻り値を期待するメッセージとそれ以外を分類できます。リスト 3-1に、戻り値を期待するAppKitのデリゲートメソッドをいくつか示します。

リスト 3-1 戻り値があるデリゲートメソッドの例

```
- (BOOL)application:(NSApplication *)sender  
    openFile:(NSString *)filename;                // NSApplication
```



```
- (BOOL)application:(UIApplication *)application
    handleOpenURL:(NSURL *)url;                                // UIApplicationDelegate

- (UITableViewIndexSet *)tableView:(NSTableView *)tableView
    willSelectRows:(UITableViewIndexSet *)selection;          // UITableViewDelegate

- (CGRect)windowWillUseStandardFrame:(NSWindow *)window
    defaultFrame:(CGRect)newFrame;                             // NSWindow
```

デリゲートは、上記のメソッドを実装することにより、発生しようとしているイベントをブロックする（第1、第2のメソッドでNOを返す）、あるいは示された値（第3のメソッドにおけるインデックス群、第4のメソッドにおける枠の矩形）を変更することができます。さらに、発生しようとしているイベントを遅延させることも可能です。たとえば、デリゲートにapplicationShouldTerminate:メソッドを実装し、NSTerminateLaterを返すことにより、アプリケーションの停止を先に延ばすことができます。

戻り値を期待しない（戻り値型がvoid）メッセージにより起動されるデリゲートメソッドもあります。このメッセージは純粋に情報を提供するためのものであり、メッセージ名には多くの場合、「Did」や「Will」など、イベントがすでに発生した、あるいは発生しようとしていることを表す字句が含まれています。リスト 3-2にこの種のデリゲートメソッドの例を示します。

### リスト 3-2 voidを返すデリゲートメソッドの例

```
- (void) tableView:(NSTableView*)tableView
    mouseDownInHeaderOfTableColumn:(NSTableColumn *)tableColumn;    // NSTableView

- (void)windowDidMove:(NSNotification *)notification;                // NSWindow

- (void)application:(UIApplication *)application
    willChangeStatusBarFrame:(CGRect)newStatusBarFrame;              //
UIApplication

- (void)applicationWillBecomeActive:(NSNotification *)notification; //
NSApplication
```

ここに挙げたメソッド群について、注意すべき点が2つあります。まず、（第3のメソッドのように）「Will」という助動詞がついていても、戻り値が期待されているとは限らない、ということです。この場合、イベントは今まさに起こるところであってブロックできませんが、デリゲートにはイベントを処理するために準備する機会が与えられます。

もうひとつは、リスト 3-2に挙げた第2と第4のメソッドに関するものです。メソッドの唯一の引数がNSNotificationオブジェクトであり、したがってこのメソッドは、ある種の通知をポストした結果として起動されます。たとえばwindowDidMove:メソッドは、NSWindowの通知NSWindowDidMoveNotificationに対応します。AppKitにおいては、通知とデリゲートメッセージと

の関係を理解することが大切です。デリゲートオブジェクトは自動的にそのデリゲートを、自身がポストする通知すべてのオブザーバとします。デリゲートはすべて、この通知を取得するための、対応するメソッドを実装しなければなりません。

カスタムクラスのインスタンスをAppKitオブジェクトのデリゲートにするためには、Interface Builder上で、このインスタンスをdelegateというアウトレットまたはプロパティに接続するだけで充分です。デリゲートオブジェクトのsetDelegate:メソッドまたはdelegateプロパティを使って、同じ設定をプログラムで行うことも可能です。できるだけ早期に、たとえばawakeFromNibやapplicationDidFinishLaunching:メソッドの中で設定します。

## デリゲートとアプリケーションフレームワーク

Cocoa / Cocoa Touchアプリケーションのデリゲートオブジェクトは、多くの場合、UIApplication、NSWindow、NSTableViewなどのレスポنداオブジェクトです。デリゲートオブジェクト自身は一般に、アプリケーションのある部分を制御するオブジェクト（すなわち調整型コントローラオブジェクト）、しかも多くの場合はカスタムオブジェクトです（もっとも、常にそうとは限りません）。次のAppKitクラスにはデリゲートが定義されています。

- NSApplication
- NSBrowser
- NSControl
- NSDrawer
- NSFontManager
- NSFontPanel
- NSMatrix
- NSOutlineView
- NSSplitView
- NSTableView
- NSTabView
- NSText
- NSTextField
- NSTextView
- NSWindow

UIKitフレームワークでもデリゲートは多用されており、いずれも公式プロトコルで実装しています。アプリケーションデリゲートは、iOS用アプリケーションにおいては非常に重要です。アプリケーションの起動や停止、メモリ不足など、アプリケーションオブジェクトからのさまざまなメッセージに回答しなければならないからです。アプリケーションデリゲートは、UIApplicationDelegateプロトコルに従わなければなりません。

デリゲートオブジェクトはデリゲートを、「リテン(retain、保持)」はしません（するべきではありません）。しかし、デリゲートオブジェクトのクライアント（通常はアプリケーション）は、デリゲートメッセージをデリゲートが受信できるようにする責任を負います。そのためには、デリゲートをメモリマネージドコードにリテンする必要があるかも知れません。この点に注意する必要があることは、データソース、通知オブザーバ、アクションメッセージのターゲットについても同様です。なお、ガベージコレクション機能が組み込まれた環境では、リテンサイクル(循環参照)の問題が起らないので、デリゲートへの参照は強い参照であることに注意してください。

AppKitのクラスには、モーダルデリゲートという、さらに制限された種類のデリゲートを持つものがあります。該当するクラス（たとえばNSOpenPanel）のオブジェクトは、モーダルダイアログを開き、「OK」ボタンをユーザがクリックしたとき、指定されたデリゲートのハンドラメソッドを起動するようになっています。モーダルデリゲートはモーダルダイアログの操作中のみ有効です。

## フレームワーククラスのデリゲートになること

デリゲートを実装したフレームワーククラスやその他のクラスには、delegateプロパティとプロトコル（通常は公式プロトコル）が宣言されています。プロトコルには、デリゲートに実装すべきメソッド（必須、省略可とも）が示されています。フレームワークオブジェクトのデリゲートとして機能するクラスのインスタンスには、次の機能が必要です。

- オブジェクトを（delegateプロパティに代入して）デリゲートとして設定する。プログラムで実行しても、Interface Builder上で設定しても構いません。
- プロトコルが公式のものであれば、クラスがこれに準拠している旨をクラス定義に宣言する。たとえば次のように宣言します。

```
@interface MyControllerClass : UIViewController <UIAlertViewDelegate> {
```

- プロトコルの必須メソッドをすべて実装し、省略可のメソッドも必要に応じて実装する。

## delegateプロパティに基づくオブジェクトの検索

デリゲートにはほかの用途もあります。たとえば、同じプログラム中で連携する2つのコントローラが、デリゲートを介して互いに相手を見つけ、通信し合えるようになります。たとえば、アプリケーション全体を制御するオブジェクトは、そのインスペクタウインドウ（その時点におけるキーウインドウと想定）のコントローラを、次のようなコードで検索できます。

```
id winController = [[NSApp keyWindow] delegate];
```

独自に実装するコードでも同様に、アプリケーションコントローラオブジェクト（定義により、グローバルアプリケーションインスタンスのデリゲート）を、次のようなコードで検索できます。

```
id appController = [NSApp delegate];
```

## データソース

「データソース」もデリゲートに似ていますが、ユーザインターフェイスではなく、データの制御を委譲される点が異なります。データソースは、`NSView`や`UIView`オブジェクト（テーブルビュー、アウトラインビューなど）がリテインするアウトレットで、ここから取得したデータを各行に埋めて表示するようになっています。ビューのデータソースは通常、デリゲートとして振る舞うオブジェクトと同じものですが、そうでなくても構いません。デリゲートと同様、データソースにも非公式プロトコルのメソッドをいくつか実装して、必要なデータをビューに供給する必要があります。また、より高度な実装では、ビュー上でユーザが直接編集するデータの処理メソッドも実装します。

データソースもデリゲートと同様、データを要求するオブジェクトからのメッセージを受け取るために必要なオブジェクトです。データソースを利用するアプリケーションは、その永続性を保証し、必要ならばメモリマネージドコードにリテインしなければなりません。

データソースは、ユーザインターフェイスオブジェクトに渡すオブジェクトの永続性に関して責任を負います。すなわち、当該オブジェクトのメモリ管理に責任を負う、ということです。これに対し、アウトラインビュー、テーブルビューなどのビューオブジェクトは、データソースから取得したデータを、実際に使っている間保持しています。とは言え、長い期間にわたってデータを使うわけではありません。表示に必要な間しか保持していないのが普通です。

## カスタムクラスにデリゲートを実装する手順

カスタムクラスにデリゲートを実装する手順は次の通りです。

- クラスのヘッダファイルに、デリゲートのアクセサメソッドを宣言する。

```
- (id)delegate;  
- (void)setDelegate:(id)newDelegate;
```

- アクセサメソッドを実装する。メモリマネージドプログラムでは、リテインサイクルを避けるため、セッターメソッドでデリゲートを保持し、あるいはコピーすることはできません。

```
- (id)delegate {  
    return delegate;  
}  
  
- (void)setDelegate:(id)newDelegate {  
    delegate = newDelegate;  
}
```

ガベージコレクション機能が組み込まれた、リテインサイクルの問題が生じない環境では、デリゲートを (`__weak` 型修飾子を使って) 弱い参照にすることは避けなければなりません。リテインサイクルについては『*Advanced Memory Management Programming Guide*』を参照してください。また、ガベージコレクションにおける弱い参照については、『*Garbage Collection Programming Guide*』の“Garbage Collection for Cocoa Essentials”を参照してください。

- デリゲートに対する公式または非公式のプロトコル（プログラムインターフェイスを含む）を宣言する。非公式のプロトコルは、`NSObject`クラスのカテゴリに記述します。デリゲートに対する公式プロトコルを宣言する場合、省略可のメソッドに、`@optional`記述子を使って印をつけてください。

デリゲートメソッドの名前については、“[デリゲートメッセージの形式](#)”（24 ページ）を参照してください。

- デリゲートメソッドを起動する前に、デリゲートに当該メソッドが実装されているかどうか、`respondsToSelector:`メッセージを送信することにより確認する。

```
- (void)someMethod {  
    if ( [delegate respondsToSelector:@selector(operationShouldProceed)] ) {  
  
        if ( [delegate operationShouldProceed] ) {  
            // 適切な処理を実行  
        }  
    }  
}
```

この確認が必要なのは、公式プロトコルの省略可のメソッド、または非公式プロトコルのメソッドについてです。

# イントロスペクション

イントロスペクションはオブジェクト指向言語/環境で利用できる強力な機能であり、Objective-CやCocoaにおいてもその例外ではありません。これは、実行時に、オブジェクト自身の内部の詳細を調べる機能のことです。具体的には、継承ツリーにおける当該オブジェクトの位置、あるプロトコルに準拠しているか否か、あるメッセージに応答できるか否か、などを調べることができます。NSObjectプロトコルおよびクラスには、実行時にオブジェクトの特性を調べるための、さまざまなイントロスペクションメソッドが定義されています。

イントロスペクションは、うまく使えば、オブジェクト指向プログラムがより効率的かつ頑健になります。メッセージのディスパッチ処理失敗、オブジェクトの等価性に関する想定への誤り、その他の問題を回避するために役立ちます。以下の各節では、NSObjectのイントロスペクションメソッドを、コード中で効果的に利用する方法を解説します。

## 継承関係の評価

オブジェクトが属するクラスが分かれば、当該オブジェクトについて、かなりの情報が得られたことになります。その能力や属性、応答できるメッセージなどが分かるのです。オブジェクトが属するクラスの細かい仕様が分からない場合であっても、誤って不適切なメッセージを送信することは避けることができます。

NSObjectプロトコルには、オブジェクトがクラス階層上のどの位置にあるか、を判断するためのメソッドがいくつか宣言されています。各メソッドは、それぞれ異なる粒度で情報を返すようになっています。たとえばインスタンスメソッドclassおよびsuperclassは、レシーバが属するクラスやスーパークラスを表す、Classオブジェクトを返します。このメソッドを活用するためには、Classオブジェクトどうしを比較する手段が必要です。リスト4-1に、単純な（一見つまらない）使い方の例を示します。

リスト 4-1 classメソッド、superclassメソッドの使い方

```
// ...
while ( id anObject = [objectEnumerator nextObject] ) {
    if ( [self class] == [anObject superclass] ) {
        // 適切な処理を実行...
    }
}
```

```
}
```

**注意** classメソッドやsuperclassメソッドは、クラスメッセージの適切なレシーバを取得するために使うこともあります。

より一般に、あるオブジェクトが属するクラスを、継承関係を含めて確認したい場合は、isKindOfClass:メッセージ、あるいはisMemberOfClass:メッセージを使います。isKindOfClass:メッセージは、レシーバがあるクラス、またはこれを継承する子孫クラスの、インスタンスであるか否かを返します。一方、isMemberOfClass:メッセージは、レシーバが特定のクラスのインスタンスであるか否かを返します。一般にisKindOfClass:メソッドの方が役に立つでしょう。というのも、当該オブジェクトに送信できるメッセージを、一括して調べることができるからです。リスト 4-2のコード断片を考えてみましょう。

#### リスト 4-2 isKindOfClass:の使い方

```
if ([item isKindOfClass:[NSData class]]) {  
    const unsigned char *bytes = [item bytes];  
    unsigned int length = [item length];  
    // ...  
}
```

オブジェクト`item`がNSDataクラスから継承している旨が分かれば、NSDataのbytesメッセージ、lengthメッセージを送信できることになります。isKindOfClass:とisMemberOfClass:の違いが現れるのは、`item`がNSMutableDataのインスタンスである場合です。isKindOfClass:をisMemberOfClass:で置き換えたとすれば、条件が真になった場合のブロックは決して実行されません。`item`はNSDataではなく、そのサブクラスであるNSMutableDataのインスタンスだからです。

## メソッドの実装とプロトコルへの準拠

NSObjectにはさらに2つ、より強力なイントロスペクションメソッド、respondsToSelector:およびconformsToProtocol:があります。それぞれ、オブジェクトにあるメソッドが実装されているか、ある公式プロトコルに準拠しているか（すなわち、プロトコルを採用し、そのメソッドをすべて実装しているか）を調べるために使います。

どちらも似たような状況で必要になるでしょう。どのクラスに属するか確定しないオブジェクトについて、ある一連のメッセージに適切に応答できるか否か、実際に送信する前に判定できます。事前にこの検査を行うことにより、セレクトアが認識できないために実行時例外が発生するのを避けることが



できます。AppKitフレームワークには、デリゲートの基盤である非公式プロトコルが、デリゲートにデリゲートメソッドが実装されているか、（respondsToSelector:を使って）当該メソッドを起動する前に検査する、という方法で実装されています。

リスト 4-3に、コード中でrespondsToSelector:メソッドを使う例を示します。

**リスト 4-3** respondsToSelector:の使用例

```
- (void)doCommandBySelector:(SEL)aSelector {
    if ([self respondsToSelector:aSelector]) {
        [self performSelector:aSelector withObject:nil];
    } else {
        [_client doCommandBySelector:aSelector];
    }
}
```

リスト 4-4に、コード中でconformsToProtocol:メソッドを使う例を示します。

**リスト 4-4** conformsToProtocol:の使用例

```
// ...
if (!([((id)testObject) conformsToProtocol:@protocol(NSMenuItem]))) {
    NSLog(@"Custom MenuItem, '%@'", not loaded; it must conform to the
        'NSMenuItem' protocol.\n", [testObject class]);
    [testObject release];
    testObject = nil;
}
```

## オブジェクトの比較

hashメソッドやisEqual:メソッドは、厳密にはイントロスペクションメソッドではありませんが、実質的に同じような役割を果たします。オブジェクトを識別しあるいは比較するために不可欠の実行時ツールと言えるでしょう。しかし、（イントロスペクションのように）オブジェクトに関する情報を実行時に調べるのではなく、クラス特有の比較ロジックを利用しています。



hashもisEqual:も、NSObjectプロトコルに宣言されており、相互に関連するメソッドです。hashメソッドは、オブジェクトをハッシュテーブルに登録する際の、テーブルアドレス（インデックス）として使う整数を返すよう実装しなければなりません。2つのオブジェクトが（isEqual:メソッドで判定した結果）等しければ、ハッシュ値も等しくなければなりません。オブジェクトをNSSetなどのコレクションオブジェクトに登録する場合、hashを定義し、2つのオブジェクトが等しければ同じハッシュ値を返す、という不変表明を満たすようにする必要があります。NSObjectにおけるisEqual:のデフォルト実装では、単にポインタ値を比較しています。

isEqual:メソッドの使い方は明らかでしょう。レシーバと、引数として渡されたオブジェクトを比較します。オブジェクトに対する処理を実行時に判断するため、比較が必要になることは少なくありません。リスト 4-5のように、あるアクション（この例では修正した環境設定の保存）を実行するかどうか、isEqual:を使って判断することができます。

#### リスト 4-5 isEqual:の使用例

```
- (void)saveDefaults {
    NSDictionary *prefs = [self preferences];
    if (![origValues isEqual:prefs])
        [Preferences savePreferencesToDefaults:prefs];
}
```

サブクラスを作成する際には、isEqual:をオーバーライドし、等価性の検査項目を追加する必要があります。あるかも知れませんが、2つのインスタンスが等しいと言うためには、サブクラスで追加した属性の値も等しくなければならない、ということがあるからです。たとえば、NSObjectのサブクラスであるMyWidgetでは、nameおよびdataというインスタンス変数を宣言しているとしましょう。MyWidgetの2つのインスタンスが等しいと言うためには、この2つのインスタンス変数の値が等しくなければなりません。リスト 4-6に、MyWidgetクラスのisEqual:の実装例を示します。

#### リスト 4-6 isEqual:のオーバーライド

```
- (BOOL)isEqual:(id)other {
    if (other == self)
        return YES;
    if (![other || ![other isKindOfClass:[self class]]])
        return NO;
    return [self isEqualToWidget:other];
}

- (BOOL)isEqualToWidget:(MyWidget *)aWidget {
```

```
if (self == aWidget)
    return YES;
if (![id][self name] isEqual:[aWidget name])
    return NO;
if (![self data] isEqualToData:[aWidget data])
    return NO;
return YES;
}
```

この`isEqual:`メソッドは、まずポインタを比較し、次にクラスの等価性を調べ、最後にオブジェクトの比較メソッドを起動しています。この比較メソッドの名前に、比較相手のクラス名が含まれていることに注意してください。**Cocoa**の比較メソッドではこのように、まず渡されたオブジェクトの型を検査することが共通の規約になっています。たとえば、`NSString`クラスの`isEqualToString:`メソッド、`NSTimeZone`クラスの`isEqualToTimeZone:`メソッドがこの規約に従っています。クラスに特有の比較メソッド（この例では`isEqualToWidget:`）が、`name`と`data`が等しいかどうかを検査します。

**Cocoa**フレームワークには「`isEqualToType:`」という形の名前のメソッドが多数ありますが、いずれも`nil`を有効な引数として認めず、`nil`を受け取った場合は例外を投げるように実装されています。ただし、後方互換性を維持するため、`isEqual:`メソッドは`nil`を受け付け、`NO`を返すようになっています。

# オブジェクトの割り当て

オブジェクトを割り当てる際に何が起きているか、その一部は「割り当て」という言葉からおおよそ見当がつくでしょう。Cocoaはアプリケーション仮想メモリ領域から、オブジェクトを格納するために十分な量のメモリを割り当てます。この容量を計算するためには、該当するクラスに定義されたインスタンス変数（型や順序を含む）考慮しなければなりません。

オブジェクトの割り当ては、そのクラスに`alloc`メッセージまたは`allocWithZone:`メッセージを送って行います。戻り値は当該クラスの「生の」（初期化していない）インスタンスです。`alloc`メソッドは、アプリケーションのデフォルトゾーンからメモリを確保します。ゾーンとは、ページ境界で区切られたメモリ領域で、アプリケーションは関連するオブジェクトやデータをこの中に保持します。ゾーンについて詳しくは、『*Advanced Memory Management Programming Guide*』を参照してください。

割り当てメッセージは、メモリを割り当てるほかにも重要な処理をしています。

- オブジェクトのリテインカウントを1と設定する。
- オブジェクトの`isa`インスタンス変数が、当該オブジェクトが属するクラス（クラス定義に基づき生成される独立した実行時オブジェクト）を指すよう初期化する。
- ほかのインスタンス変数を0（またはそれぞれの型の0に相当する値、`nil`、`NULL`、`0.0`など）に初期化する。

`isa`インスタンス変数は`NSObject`から継承したものであり、Cocoaオブジェクトすべてに共通です。属するクラスを`isa`に設定した後、オブジェクトは、継承階層の実行時ビュー、およびプログラムを構成するオブジェクト（クラスおよびインスタンス）のネットワークに組み込まれます。その結果、オブジェクトは、実行時に必要な情報をすべて調べられるようになります。継承階層におけるほかのオブジェクトの位置、ほかのオブジェクトが準拠するプロトコル、メッセージに応答して実行できるメソッドが実装されている場所などです。

以上のように、「割り当て」では単にオブジェクト用のメモリ領域を割り当てるだけでなく、小さいけれどもどのオブジェクトにとっても非常に重要な2つの属性、すなわち`isa`インスタンス変数とリテインカウントを初期化します。さらに、それ以外のインスタンス変数を0にします。しかし、結果として得られるオブジェクトは、まだそのままでは使えません。`init`などの初期化メソッドで、それぞれの特性に合わせて初期化することにより、正常に機能するオブジェクトが得られるのです。

# オブジェクトの初期化

初期化の処理では、オブジェクトのインスタンス変数を、合理的かつ有用な初期値に設定します。また、オブジェクトが必要とするほかのグローバルリソースを割り当て、準備し、必要に応じてファイルなどの外部リソースから読み込む、といった処理も可能です。インスタンス変数を宣言しているオブジェクトには、初期化メソッドを実装しなければなりません（すべてデフォルト値、すなわち0で構わない場合を除く）。イニシャライザ（initializer）が実装されていなければ、Cocoaは代わりに、最も近い祖先のイニシャライザを呼び出します。

## イニシャライザの形式

NSObjectにはイニシャライザのプロトタイプとしてinitが宣言されています。これはインスタンスメソッドで、戻り値はid型のオブジェクトです。初期化に際して何らかのデータを渡す必要がないのであれば、initをオーバーライドするとよいでしょう。しかし、オブジェクトを合理的な初期状態にするために、外部データを必要とすることも少なくありません。たとえばAccountクラスを考えてみましょう。Accountオブジェクトを適切に初期化するためには、一意的な口座番号が必要であり、これをイニシャライザに渡さなければなりません。このようにイニシャライザは、引数をいくつか取ることがあります。初期化メソッドに対する要件は、その名前が「init」で始まることだけです（そのため、一般にイニシャライザを表す場合、「init...」と表記する規約になっています）。

---

**注意** サブクラスに、引数を取るイニシャライザを実装する代わりに、引数なしのinitメソッドのみ実装しておき、初期化後に「設定」アクセサメソッドで適切な初期状態にする、という方法もあります（アクセサメソッドは、インスタンス変数の値を設定/取得する手段であり、これによりオブジェクトデータのカプセル化を実現します）。あるいは、サブクラスにプロパティを定義し、そのアクセス構文が使えるようにしている場合は、初期化後にプロパティ値を設定する、という方法も考えられます。

---

Cocoaには引数付きのイニシャライザの例が多数あります。その例をいくつか示します（括弧内にクラスを注記）。

- (id)initWithArray:(NSArray \*)array; (NSSetから)
- (id)initWithTimeInterval:(NSTimeInterval)secsToBeAdded sinceDate:(NSDate \*)anotherDate; (NSDateから)

- (id)initWithContentRect:(CGRect)contentRect styleMask:(unsigned int)aStyle backing:(NSBackingStoreType)bufferingType defer:(BOOL)flag; (NSWindowから)
- (id)initWithFrame:(CGRect)frameRect; (NSControlおよびNSViewから)

以上のイニシャライザは、いずれも名前が「init」で始まるインスタンスメソッドで、動的な型であるid型のオブジェクトを返します。また、複数の引数を取る場合は、Cocoaの規約に従い、最も重要な第1引数の前に、「WithType:」や「FromSource:」という形の字句を置いています。

## イニシャライザに関する問題

「init...」メソッドは、そのシグニチャに従い、オブジェクトを戻り値とする必要があります。しかしそのオブジェクトは、直前に割り当てられたもの（init...メッセージのレシーバ）でなくても構いません。言い替えると、イニシャライザの戻り値であるオブジェクトが、このとき初期化されたものであるとは限らないのです。

このように、今まさに割り当てたオブジェクト以外のものを返す、という状況は2通り考えられます。1つめはさらに、関連する2つの状況に分けられます。オブジェクトがシングルトンインスタスでなければならない場合と、オブジェクトのある属性値が一意でなければならない場合です。NSWorkspaceなどいくつかのCocoaクラスは、プログラム全体で1つのインスタンスしか存在できません。このような場合、インスタンスが高々1つであることを保証し（イニシャライザでも行えるが、クラスファクトリメソッドで実装する方が望ましい）、新たにインスタンスを生成するよう要求されても、この唯一のインスタンスを返すようにする必要があります。

同じような状況は、オブジェクトのある属性値が一意でなければならない場合にも当てはまります。先に例を挙げたAccountクラスを考えてみましょう。口座には一意的な口座番号が必要です。このクラスのイニシャライザ、たとえばinitWithAccountID:に引数として渡された口座番号が、ほかのオブジェクトに対して設定済みのものであった場合、次の2つの処理が必要です。

- いったん割り当てたオブジェクトを（メモリマネージドコードで）解放する
- 同じ口座番号で初期化済みのAccountオブジェクトを返す

これにより、口座番号の一意性を維持しつつ、指定された口座番号のAccountインスタンスを返すことができます。

2つめの状況を考えてみましょう。「init...」メソッドが要求された通りの初期化を行えない場合です。たとえばinitWithFile:メソッドは、引数として指定されたパスのファイルを読み込み、それに従ってオブジェクトを初期化することになっています。しかし、該当するファイルがなかった場合、初期化はできません。同様の問題は、たとえばイニシャライザinitWithArray:に、NSArrayではなくNSDictionaryのオブジェクトが渡された場合にも起こります。「init...」メソッドがオブジェクトを初期化できない場合、次のような処理が必要です。

- いったん割り当てたオブジェクトを（メモリマネージドコードで）解放する
- `nil`を返す

イニシャライザの戻り値が`nil`であれば、要求されたオブジェクトが生成できなかったことを表します。一般に、オブジェクトを生成したときは、戻り値が`nil`でないかどうか確認してから次の処理に進むようにしてください。

```
id anObject = [[MyClass alloc] init];
if (anObject) {
    [anObject doSomething];
    // 続きの処理...
} else {
    // エラー処理
}
```

`init...`メソッドは、`nil`、あるいは明示的に割り当てた以外のオブジェクトを返すこともあるので、イニシャライザではなく`alloc`や`allocWithZone:`から返されたインスタンスを使うのは危険です。たとえば次のコードを考えてみましょう。

```
id myObject = [MyClass alloc];
[myObject init];
[myObject doSomething];
```

この例で、`init`メソッドの戻り値は、`nil`、あるいはほかのオブジェクトに置き換えられている可能性があります。`nil`にメッセージを送っても例外は発生しないので、表面上は何も起こらないかも知れません（デバッグには苦勞しそうですが）。しかし、割り当てただけの「生の」インスタンスではなく、初期化済みのインスタンスを常に使うようにする必要があります。したがって、割り当てメッセージ（`alloc`）を初期化メッセージ（`init`）内に入れ子にし、イニシャライザから返されたオブジェクトをテストした上で、次の処理に移るようにしてください。

```
id myObject = [[MyClass alloc] init];
if ( myObject ) {
    [myObject doSomething];
} else {
    // エラーからの回復...
}
```

初期化済みのオブジェクトを、重ねて初期化することはできません。再初期化を試みると、多くの場合、当該オブジェクトが属するフレームワーククラスから例外が発生します。たとえば次の例では、2回目の初期化の際に`NSInvalidArgumentException`が発生します。

```
NSString *aStr = [[NSString alloc] initWithString:@"Foo"];  
aStr = [aStr initWithString:@"Bar"];
```

## イニシャライザの実装

`init...`メソッドを実装する際には、いくつかある重要な規約に従わなければなりません。クラスの唯一のイニシャライザである場合のほか、複数ある場合の**指定イニシャライザ**（**designated initializer**、**“複数のイニシャライザ、指定イニシャライザ”**（42 ページ）を参照）についても同じです。

- スーパークラス（`super`）のイニシャライザを、実装の先頭で起動してください。
- スーパークラスのイニシャライザから返されたオブジェクトを検査します。`nil`であれば初期化を続行できません。レシーバに`nil`を返してください。
- オブジェクトの参照であるインスタンス変数を初期化する際には、必要に応じ、オブジェクトのコピーを（メモリマネージドコードで）リテインしてください。
- インスタンス変数に初期化を設定した後、次の場合を除き、`self`を返します。
  - 代替のオブジェクトを返す必要がある場合。この場合はまず、直前に割り当てたオブジェクトを（メモリマネージドコードで）解放します。
  - 問題が発生して初期化を続行できなかった場合。この場合は`nil`を返します。

```
- (id)initWithAccountID:(NSString *)identifier {  
    if ( self = [super init] ) {  
        Account *ac = [accountDictionary objectForKey:identifier];  
        if (ac) { // 同じ口座番号のオブジェクトがすでに存在する  
            [self release];  
            return [ac retain];  
        }  
        if (identifier) {  
            accountId = [identifier copy]; // accountIdはインスタンス変数  
            [accountDictionary setObject:self forKey:identifier];  
            return self;  
        } else {  

```

```
        [self release];  
        return nil;  
    }  
    } else  
        return nil;  
}
```

---

**注意** 簡単にするため、この例では引数がnilのとき単にnilを返していますが、Cocoaの場合、例外を投げる方が優れています。

---

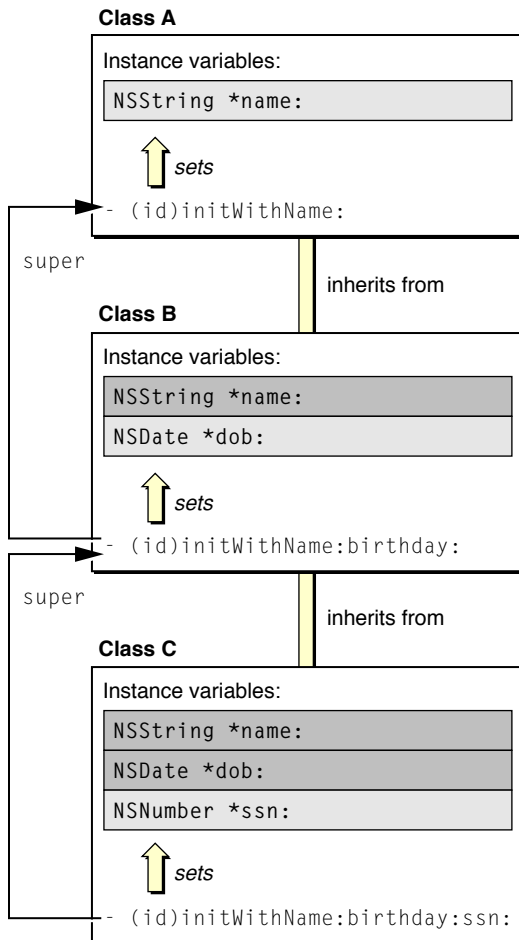
オブジェクトのインスタンス変数を、すべて明示的に初期化しなくても構いません。正常に機能するために必要な範囲でよいのです。割り当ての際に0で初期化されるため、それだけで十分なことも少なくありません。メモリ管理のために必要なので、インスタンス変数は確実にリテインまたはコピーしてください。

スーパークラスのイニシャライザを実装の先頭で起動する、という要件は重要です。オブジェクトのカプセル化は、当該クラスで定義されたインスタンス変数だけでなく、祖先クラスすべてのインスタンス変数が対象なのでした。superのイニシャライザをまず起動することにより、継承チェーンをたどって、祖先クラスで定義されているインスタンス変数を確実に初期化できます。すぐ上のスーパークラスは、そのイニシャライザの中でさらにそのスーパークラスのイニシャライザを起動します。これがさらに、スーパークラスのinit...メソッドを起動する、という具合に順次さかのぼって行くの



です（図6-1を参照）。初期化の順序も重要です。サブクラス側の初期化を後で行うことにより、スーパークラスで定義されたインスタンス変数が、合理的な値に初期化されていることを前提にして処理できるのです。

図 6-1 継承チェーンをさかのぼって初期化



継承したイニシャライザは、サブクラスを作成する際にも問題になります。場合によっては、スーパークラスのinit...メソッドだけで、サブクラスのインスタンスの初期化が済んでしまうこともあります。しかしこれで足りることは少ないので、スーパークラスのイニシャライザはオーバーライドするべきです。そうしないと、スーパークラスの実装が起動されてしまいますが、スーパークラス側ではサブクラスのことを意識せずに処理するので、インスタンスが正しく初期化されない恐れがあります。

## 複数のイニシャライザ、指定イニシャライザ

同じクラスに複数のイニシャライザを定義することも可能です。初期化すること自体は同じでも呼び出し形式（シグニチャ）が異なるので、呼び出し側では状況に応じて使い分けることができます。たとえばNSSetクラスにはイニシャライザがいくつかあり、初期化に用いるデータの受け取り方（引数）が違います。NSArrayオブジェクトを受け取るもの、要素のリストと要素数を受け取るもの、可変長引数の形で要素を受け取る（末尾をnilで示す）ものなどです。

```
- (id)initWithArray:(NSArray *)array;
- (id)initWithObjects:(id *)objects count:(unsigned)count;
- (id)initWithObjects:(id)firstObj, ...;
```

サブクラスによっては、呼び出し側の便宜のため、引数を減らした簡易イニシャライザを用意していることがあります。省略した引数についてはデフォルト値を設定し、すべての引数を明示的に指定する「完全な」イニシャライザを呼び出します。通常これは、クラスで最も重要なイニシャライザ、すなわち指定イニシャライザです。たとえば、Taskというクラスに、次のような指定イニシャライザが宣言されているとしましょう。

```
- (id)initWithTitle:(NSString *)aTitle date:(NSDate *)aDate;
```

Taskクラスではほかに、二次（簡易）イニシャライザを用意しているかも知れません。省略された引数についてはデフォルト値を使って、指定イニシャライザを起動するだけのイニシャライザです。指定イニシャライザと二次イニシャライザの例を示します。

```
- (id)initWithTitle:(NSString *)aTitle {
    return [self initWithTitle:aTitle date:[NSDate date]];
}

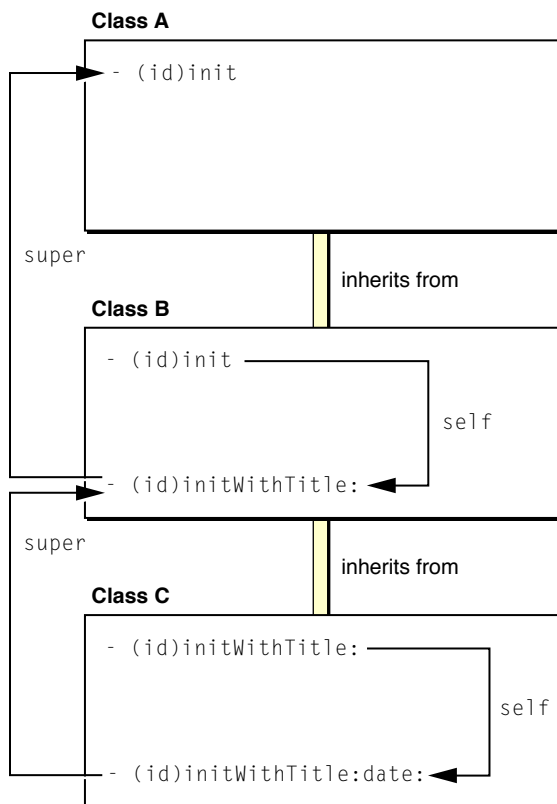
- (id)init {
    return [self initWithTitle:@"Task"];
}
```

指定イニシャライザはクラスにおいて重要な役割を担います。継承したインスタンス変数が適切に初期化されるよう、スーパークラスの指定イニシャライザを呼び出しています。また、一般にinit...メソッドの中で引数が最も多く、最も幅広い初期化処理を行います。二次イニシャライザは、selfをレシーバとし、指定イニシャライザを起動する形で実装します。

サブクラスを定義する際には、スーパークラスの指定イニシャライザがどれであることを知らなければなりません。サブクラス側の指定イニシャライザでは、`super`をレシーバとして、この指定イニシャライザを起動することになります。また、継承したほかのイニシャライザも同様に扱う必要があります。簡易イニシャライザも必要に応じて定義してください。クラスのイニシャライザを設計する際には、`super`を介し、継承階層をたどって指定イニシャライザを連鎖させることが重要です。それ以外のイニシャライザは、`self`を介して、当該クラスの指定イニシャライザに連鎖させます。

以上の説明は、例を見た方が分かりやすいでしょう。たとえば、A、B、Cという3つのクラスを考えます。BはAのサブクラス、CはBのサブクラスです。各サブクラスは、属性をインスタンス変数として追加し、これを初期化する`init...メソッド`（指定イニシャライザ）を実装しています。さらに、二次イニシャライザを定義し、継承したイニシャライザは必要に応じてオーバーライドしています。図6-2に、3つのクラスのイニシャライザとそれぞれの関係を示します。

図 6-2 二次イニシャライザと指定イニシャライザの関係



各クラスの指定イニシャライザは、最も幅広い初期化処理を行うイニシャライザであり、サブクラスで追加した属性はここで初期化します。指定イニシャライザもやはり`init...メソッド`であり、`super`に対するメッセージの形で、スーパークラスの指定イニシャライザを呼び出します。この例で、クラスCの指定イニシャライザである`initWithTitle:date:`は、そのスーパークラスBの指定イニシャ

イザである `initWithTitle:` を呼び出しています。これはさらに、クラスAの `init` メソッドを呼び出します。サブクラスを定義する際には、スーパークラスの指定イニシャライザがどれであるかを知ることが重要です。

指定イニシャライザはこのように、`super` へのメッセージという形で継承チェーンを上にとどりますが、二次イニシャライザは `self` へのメッセージという形で、当該クラスの指定イニシャライザを呼び出します。二次イニシャライザの中には、（この例のように）継承したイニシャライザをオーバーライドしたものもあります。クラスCでは `initWithTitle:` をオーバーライドし、デフォルトの日付を渡して指定イニシャライザを呼び出しています。一方、この指定イニシャライザは、クラスBの指定イニシャライザ（オーバーライドする元の `initWithTitle:`）を呼び出しています。同じように `initWithTitle:` メッセージを送っても、送り先オブジェクトが属するクラスがBかCによって、実際に起動されるメソッドの実装は異なります。一方、クラスCで `initWithTitle:` をオーバーライドしていないとすれば、クラスCのインスタンスにメッセージを送っても、実際に起動されるのはクラスBの実装です。したがってCのインスタンスは、日付が初期化されず、不完全な状態になってしまいます。このように、サブクラスを定義する際には、継承したイニシャライザをすべて適切に取り扱うことが重要なのです。

場合によっては、スーパークラスの指定イニシャライザだけで充分であり、サブクラス側で別に指定イニシャライザを実装する必要がないこともあります。また、あるクラスの指定イニシャライザが、スーパークラスのそれをオーバーライドしたものであることもあります。これは、サブクラスで独自のインスタンス変数を追加していない（あるいは追加していても明示的な初期化が必要ない）けれども、スーパークラスの指定イニシャライザの処理を、サブクラス側で補う必要がある、という状況です。

# Model-View-Controllerパターン

「Model-View-Controller」（MVC）は非常に古くからあるデザインパターンです。Smalltalkが開発された時代以降、さまざまな変形が生まれてきました。アプリケーション全体のアーキテクチャに関わる、という意味で高レベルのパターンであり、アプリケーションの中で担う役割に応じてオブジェクトを分類します。さらに、基本的なパターンをいくつか組み合わせた、複合パターンであるとも言えます。

オブジェクト指向プログラムは、MVCデザインパターンに当てはめることにより、さまざまな利点が生れます。このようなプログラムのオブジェクトは、再利用性に優れ、インターフェイスもきちんと定義されている傾向があります。プログラムは一般に、要求の変化に適応しやすい、すなわち、そうでないプログラムに比べて拡張が容易であると言えます。さらに、バインディング、ドキュメントアーキテクチャ、スクリプト機能など、Cocoaのさまざまな技術やアーキテクチャがMVCを基盤としているので、カスタムオブジェクトもMVCで定義された何らかの役割を担う必要があります。

## MVCオブジェクトの役割と関係

MVCデザインパターンでは、モデル、ビュー、コントローラという3種類のオブジェクト群を考えます。アプリケーションにおいてこの3種類のオブジェクト群が果たすべき役割と、それぞれを結ぶ通信路を定義しているのです。アプリケーション設計においては、この3種類それぞれに該当するオブジェクトを選択する（あるいはカスタムクラスを作成する）、という重要な手順があります。各オブジェクト群は、概念上の（抽象的な）境界で分離し、この境界を越えて相互のやり取りを行うことになります。

## モデルオブジェクトはデータや基本的な振る舞いをカプセル化

モデルオブジェクトは、アプリケーションを特徴づける情報や技術を表します。アプリケーションが扱うデータを保持し、これを操作するロジックを定義しているのです。適切に設計されたMVCアプリケーションでは、重要なデータをモデルオブジェクトにカプセル化しています。アプリケーションの永続的な状態を構成するデータは、ファイルに保存するか、データベースに格納するかに関わらず、アプリケーションに読み込んだ後はモデルオブジェクトの管理下に置かなければなりません。ある特定の問題領域に関する情報や技術を表すものなので、さまざまな用途に使える傾向があります。

理論上、モデルオブジェクトには、これを目に見える形に表現し編集できるようにするユーザインターフェイスとの間に、明示的な接続がありません。たとえば、（住所録アプリケーションの）人を表すモデルオブジェクトには、誕生日を格納しようとするでしょう。モデルオブジェクト「Person」に格納するデータとしてふさわしいと言えます。しかし、書式文字列その他、日付の表現方法に関する情報は、恐らく別の場所で管理すべきです。

現実には、分離する方法が常に最善とは限らず、ある程度柔軟に適応する余地があります。しかし一般に、モデルオブジェクトは、インターフェイスや表現の問題に関与するべきではありません。例外扱いすることが妥当である例として、描画アプリケーションがあります。モデルオブジェクトは、表示される図形を表します。図形オブジェクトは、自分自身がどのように描画されるか、を知っていなければ意味がありません。目に見える（描画される）「もの」を定義することが、このオブジェクトの「レゾネートル（存在意義）」であるからです。しかしこの場合でも図形オブジェクトは、ある特定のビューに描画されることを前提としてはならず、したがって、自分自身をどこに描画するか、という情報は管理しません。描画を主導するビューオブジェクトに、自分自身を描画するよう依頼される、という形を取るべきなのです。

## ビューオブジェクトは情報をユーザ向けに表現

ビューオブジェクトは、モデルが表すデータをどのように表示し、どのようにすればユーザが編集できるか、その方法を知っています。当該データを保存する方法については責任を負いません（もちろん、実際にデータを保存することが決していない、という意味ではありません。性能上の理由でデータをキャッシュする、などの工夫をすることはあります）。ビューオブジェクトに表示するのは、あるモデルオブジェクトの一部だけであることも、全体であることもあります。さらに、異なる種類のモデルオブジェクトを多数表示することも可能です。ビューにもさまざまな変種があるのです。

ビューオブジェクトは一般に、再利用性が高く、さまざまな設定が可能です。また、アプリケーションをまたがって一貫性を与えることもできます。Cocoaの場合、AppKitフレームワークに多数のビューオブジェクトが定義されており、その多くはInterface Builderのライブラリとして提供されています。AppKitのビューオブジェクト、たとえばNSButtonオブジェクトをさまざまな用途に再利用することにより、どのCocoaアプリケーションのボタンも同じように振る舞うことが保証されます。したがって、アプリケーションをまたがって、外観や振る舞いの一貫性を高いレベルで保つことができるのです。

ビューはモデルを正しく表示しなければなりません。したがって、通常、モデルに変化があればそれを認識する必要があります。モデルオブジェクトは特定のビューオブジェクトに結びついていないので、変化した旨を伝える汎用的な手段が必要です。

## コントローラオブジェクトはモデルをビューに結びつける

コントローラオブジェクトは、ビューオブジェクトとモデルオブジェクトの仲介役として振る舞います。コントローラは多くの場合、ビューが表示のために必要なモデルオブジェクトにアクセスするための手段となり、また、モデルの状態変化をビューが認識するための導管として動作します。また、アプリケーションのタスク群をセットアップし調整することや、ほかのオブジェクトのライフサイクルを管理することも可能です。

Cocoaの典型的なMVC設計では、ユーザがビューオブジェクトを介して値の入力や選択を行うと、その内容がコントローラオブジェクトに伝わります。コントローラオブジェクトは、ユーザ入力をアプリケーション特有のやり方で解釈します。そしてモデルオブジェクトに、この入力を使って実行すべきこと（値を追加する、現行レコードを削除する、など）を指示し、あるいは該当するプロパティの値を変更するよう指示します。また、同じユーザ入力に基づき、ビューオブジェクトに対して、外観や振る舞いを変えるよう指示する（たとえば、ボタンに対して、押下できない状態にするよう指示する）ことも考えられます。逆にモデルオブジェクトの状態が変化した（たとえば、データソースから新たにデータを読み込んだ、など）場合も同様に、その内容がコントローラオブジェクトに伝わります。コントローラオブジェクトはこれに応じて、ビューオブジェクトに対し、表示を更新するよう指示します。

コントローラオブジェクトは、種類によって再利用性が高いものと低いものがあります。“Cocoaの各種のコントローラオブジェクト”（48ページ）に、Cocoaに組み込まれた各種のコントローラオブジェクトを説明します。

## 役割の兼務

MVCの役割を、ひとつのオブジェクトが兼務することもあります。たとえばコントローラとビューの役割を兼務するオブジェクトはビューコントローラと呼びます。同様に、モデルコントローラと呼ばれるオブジェクトもあります。アプリケーションによっては、このように役割を兼務するオブジェクトがあっても、妥当な設計として受け入れられます。

モデルコントローラは、コントローラでありながら、モデル層にも関与するオブジェクトです。これはモデルを「所有」しています。主な責務は、モデルを管理し、ビューオブジェクトと通信することです。総じて、モデルに適用されるアクションメソッドも、一般にモデルコントローラに実装します。ドキュメントアーキテクチャには、このようなメソッドが多数あります。たとえばNSDocumentオブジェクト（ドキュメントアーキテクチャの中核）は、ファイルの保存に関するアクションメソッドを自動的に処理します。

ビューコントローラは、コントローラでありながら、ビュー層にも関与するオブジェクトです。これはインターフェイス（ビュー）を「所有」しています。主な責務は、インターフェイスを管理し、モデルと通信することです。ビューに表示されるデータに関わるアクションメソッドは、一般にビューコントローラに実装します。ビューコントローラの例として、NSWindowControllerオブジェクト（これもドキュメントアーキテクチャの一端を担う）があります。



“MVCアプリケーションの設計ガイドライン”（52 ページ）に、MVCの役割を兼務するオブジェクトに関する、設計上のガイドラインをいくつか示します。

---

さらに学びたい方へ 『Document-Based Applications Overview』では、別の観点から、モデルコントローラとビューコントローラの違いを説明しています。

---

## Cocoaの各種のコントローラオブジェクト

“コントローラオブジェクトはモデルをビューに結びつける”（47 ページ）では、コントローラオブジェクトについて抽象的に概要を説明しました。しかし実際にははるかに複雑です。Cocoaには大きく分けて、仲介型（mediating）と調整型（coordinating）という2種類のコントローラオブジェクトがあります。それぞれ異なるクラス群と関連づけられており、したがってその振る舞いも違います。

仲介型コントローラは一般に、NSControllerクラスから派生したオブジェクトです。これはCocoaのバインディング技術で使われます。ビューオブジェクトとモデルオブジェクトの間を仲介し、データの受け渡しに便宜を図ります。

---

**iOSにおける注意事項** AppKitにはNSControllerとそのサブクラスが実装されています。こういったクラスやバインディング技術は、iOSでは利用できません。

---

仲介型コントローラは既製のオブジェクトであることが多く、Interface Builderライブラリからドラッグ操作により追加できます。仲介型コントローラの設定により、ビューオブジェクトとコントローラオブジェクトのプロパティ間にバインディングを確立し、次いで、コントローラのプロパティと、モデルオブジェクトのある特定のプロパティとの間に、バインディングを確立します。すると、ビューオブジェクトの表示値がユーザが変更したとき、仲介型コントローラを介して自動的に新しい値がモデルオブジェクトに伝わり、ストレージに格納されるようになります。モデルのプロパティ値が変化したときも同様に、その旨がビューに伝わり、表示にも反映されます。抽象クラスであるNSControllerと、その具象サブクラスであるNSObjectController、NSArrayController、NSUserDefaultsController、NSTreeControllerには、変更を確定または破棄する、選択値やプレースホルダ値を管理する、などの機能があります。

調整型コントローラは一般に、NSWindowControllerまたはNSDocumentControllerのオブジェクト（AppKitでのみ使用可）、あるいはNSObjectのカスタムサブクラスのインスタンスです。アプリケーションにおけるその役割は、アプリケーション全体またはその一部（nibファイルから展開（unarchive）したオブジェクトなど）の機能を、高所から見渡して調整することです。調整型コントローラは次のようなサービスを提供します。

- デリゲートメッセージに応答し、通知を監視する



- アクションメッセージに応答する
- 所有するオブジェクトのライフサイクルを管理する（適切な時点で解放するなど）
- オブジェクト間の接続を確立するなどのセットアップタスクを実行する

NSWindowControllerクラスやNSDocumentControllerクラスは、ドキュメントベースのアプリケーションを対象とするCocoaアーキテクチャの一部を成します。このクラスのインスタンスには、上記のサービスのいくつかについて、デフォルトの実装がなされています。サブクラスを作成し、アプリケーション特有の振る舞いを実装することも可能です。NSWindowControllerオブジェクトは、ドキュメントアーキテクチャを基盤としないアプリケーションのウインドウ管理にも使えます。

調整型コントローラは、`nib`ファイルにアーカイブされたオブジェクトを所有することが少なくありません。`nib`ファイルの外に位置する調整型コントローラが、ファイル所有者としての立場でそのオブジェクトを管理するのです。調整型コントローラが所有するオブジェクトには、仲介型コントローラや、ウインドウオブジェクト、ビューオブジェクトなどがあります。調整型コントローラの、ファイル所有者としての性質については、“[複合デザインパターンとしてのMVC](#)”（49 ページ）を参照してください。

NSObjectのカスタムサブクラスのインスタンスは、調整型コントローラとしても完全に適したものになりえます。このようなコントローラオブジェクトは、仲介型としての機能と調整型としての機能を併せ持っています。仲介型コントローラとして振る舞う場合には、ターゲットアクション、アウトレット、デリゲート、通知などの機構を使って、ビューオブジェクトとモデルオブジェクト間のデータの受け渡しに便宜を図ります。もっとも、グルーコードが多いという傾向があるほか、もっぱらアプリケーション特有の処理をするので、再利用性に乏しいオブジェクトと言えます。

---

さらに学びたい方へ Cocoaのバインディング技術について詳しくは、『*Cocoa Bindings Programming Topics*』を参照してください。

---

## 複合デザインパターンとしてのMVC

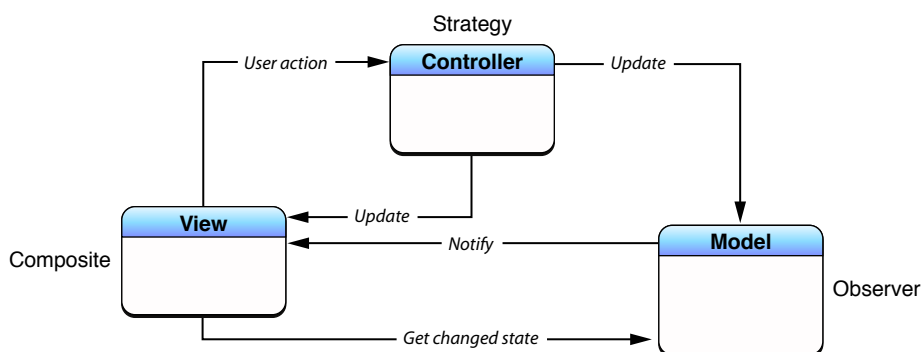
「Model-View-Controller」は、より基本的なデザインパターンをいくつか組み合わせた形のデザインパターンです。基本パターンどうしの組み合わせにより、MVCアプリケーションを特徴づける、機能の分割や通信経路を定義しています。しかし従来型のMVCは、組み合わせる基本パターンが、現在のCocoaのそれとは違っていました。その違いは主として、コントローラオブジェクトやビューオブジェクトに与える役割に関するものです。

当初の（Smalltalk流の）考え方では、MVCはComposite、Strategy、Observerというパターンから成っていました。

- **Composite**：アプリケーションのビューオブジェクトは、実際には入れ子になったビューの複合体（composite）であり、このビュー群が協調して（ビュー階層の形で）動作します。この表示コンポーネントは、ウインドウを頂点とし、その下にテーブルビューなどの複合ビュー、さらにその下にはボタンなどの分割できないビューがあります。ユーザ入力や画面表示の処理は、複合体を構成するどのレベルでも可能です。
- **Strategy**：コントローラオブジェクトは、いくつかのビューオブジェクトに対する戦略（strategy）を実装しています。ビューオブジェクトの機能は（視覚的）表示に関わる範囲に限定し、インターフェイスの振る舞いが当該アプリケーションにおいてはどのような意味を持つか、の判断はすべてコントローラに委譲します。
- **Observer**：モデルオブジェクトは、自分自身と直接的な関わり合いがあるオブジェクト（一般にビューオブジェクト）の状態が変化すると、その通知を受け取ります。

従来型のComposite-Strategy-Observerパターンは、図 7-1に示すように連携して動作します。複合体の階層のあるレベルで、ユーザがビューを操作すると、イベントが生成されます。コントローラオブジェクトはイベントを受け取り、アプリケーション特有の方法で解釈します（すなわち、戦略を適用します）。戦略には、（メッセージを介して）モデルオブジェクトに状態を変えるよう要求するもの、（複合体の階層のあるレベルで）ビューオブジェクトに振る舞いや外観を変えるよう要求するものがあります。一方、モデルオブジェクトは、状態が変化すると、オブザーバとして登録されているオブジェクトすべてにその旨を通知します。オブザーバがビューオブジェクトである場合は、通知を受けて外観を変えることも考えられます。

図 7-1 複合パターンとしてのMVC（従来型）

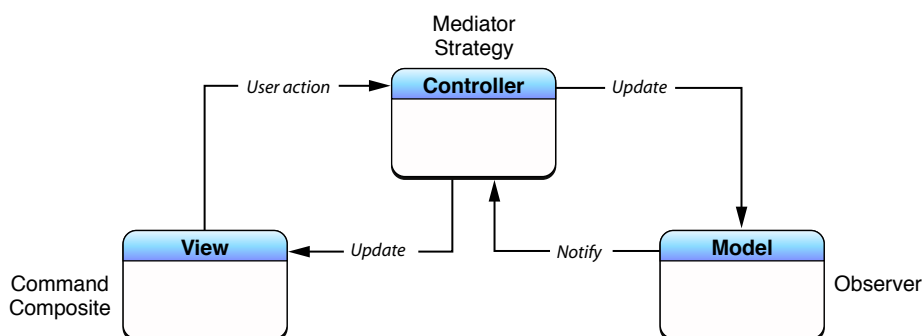


Cocoaにおける、複合パターンとしてのMVCも、ある程度これと似ています。実際、図 7-1のようにアプリケーションを構築することも、まったく問題ありません。バインディング技術を使うと、ビューがモデルオブジェクトを直接監視し、状態変化に応じて通知を受け取る形のCocoa MVCアプリケーションを、容易に構築できます。しかしこの設計には理論上の問題があります。ビューやモデルは、アプリケーションでも特に再利用性の高いオブジェクトでなければなりません。ビューオブジェクトは、オペレーティングシステムやその上で動作するアプリケーションの「look and feel（外観）」を表します。外観や振る舞いの一貫性が重要であり、再利用性が高いことが求められます。一方、モデルオ

プロジェクトはその性質上、問題領域に対応するデータをカプセル化し、そのデータに対する処理を行います。設計の観点からは、再利用性を高めるためにも、モデルとビューを完全に分離することが重要なのです。

多くのCocoaアプリケーションでは、モデルオブジェクトの状態が変化すると、その通知はコントローラオブジェクトを経由してビューオブジェクトに伝わります。図7-2にこの様子を示します。2つの基本的なデザインパターンが関与していますが、よりすっきりとしています。

図 7-2 複合パターンとしてのMVC (Cocoa)



この複合デザインパターンでは、コントローラオブジェクトはMediatorパターンとStrategyパターンを包含しています。モデルとビューの間でやり取りされるデータフローを、両方向とも仲介しています。モデルの状態変化はコントローラオブジェクトを経由してビューオブジェクトに伝わります。さらに、ビューオブジェクトには、ターゲットアクション機構の実装という形で、Commandパターンも含まれています。

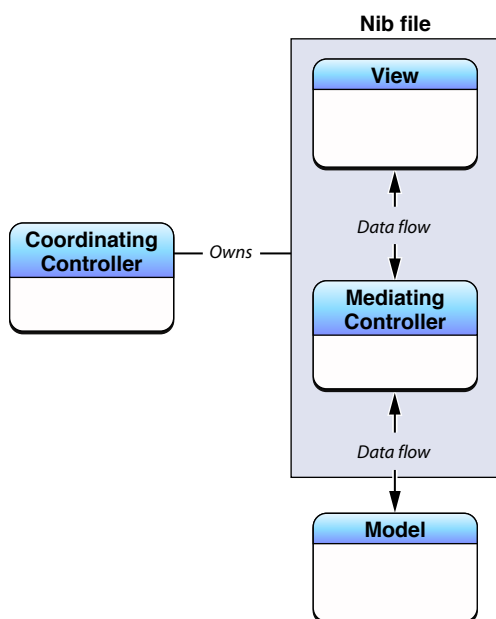
**注意** 「ターゲットアクション」は、ビューオブジェクトがユーザの入力や選択を通知するための機構であり、調整型、仲介型のどちらのコントローラオブジェクトにも実装できます。しかしこの機構の設計は、コントローラの型によって異なります。調整型コントローラの場合、Interface Builder上でビューオブジェクトをそのターゲット（コントローラオブジェクト）に接続し、所定のシグニチャに従って実装したアクションセクタを指定します。ウィンドウのデリゲートやグローバルアプリケーションオブジェクトが利用できるので、レスポンドチェーンに組み込むことも可能です。一方、仲介型コントローラに使われるバインディング機構も、ビューオブジェクトをターゲットに接続するものですが、そのアクションは引数の個数や型は自由です。ただしレスポンドチェーンに組み込むことはできません。

図7-2に示した改良型の複合デザインパターンは、特にこれがMediatorデザインパターンに基づいている、実用的にも理論的にも長所があります。仲介型コントローラはNSControllerの具象サブクラスから派生したもので、Mediatorパターンを実装しているだけでなく、選択値やプレースホルダ値の管理など、アプリケーションにとって役に立つさまざまな機能を提供します。さらに、バインディン

グ技術を使わない場合でも、ビューオブジェクトはCocoa通知センターなどの機構を使って、モデルオブジェクトから通知を受け取ることができます。しかしそのためには、ビューのサブクラスを別途作成して、モデルオブジェクトがポストする通知に関する処理を追加しなければなりません。

適切に設計されたCocoa MVCアプリケーションでは、多くの場合、調整型コントローラオブジェクトが仲介型コントローラを所有し、これはnibファイルにアーカイブされています。図 7-3に、この2種類のコントローラオブジェクトの関係を示します。

図 7-3 nibファイルの所有者としてこれを管理する調整型コントローラ



## MVCアプリケーションの設計ガイドライン

以下、アプリケーションの設計においてModel-View-Controllerを検討する際のガイドラインをいくつか示します。

- NSObjectのカスタムサブクラスのインスタンスを仲介型コントローラとして使うことも可能ですが、そのために必要な処理を、すべて独自に実装する必要はありません。Cocoaバインディング技術を想定して設計された、既製のNSControllerオブジェクトを使うとよいでしょう。具体的には、NSControllerの具象サブクラスである、NSObjectController、NSArrayController、NSUserDefaultsController、NSTreeControllerのいずれか、またはそのカスタムサブクラスのインスタンスを使ってください。

しかし、アプリケーションが非常に単純であり、若干のグルーコードを記述するだけで、アウトレットやターゲットアクションを使って仲介役としての振る舞いを実装できるのであれば、NSObjectのカスタムサブクラスのインスタンスを仲介型コントローラとして使っても構いません。NSObjectのカスタムサブクラスでは、仲介型コントローラをNSControllerのように、キー値コーディング、キー値監視、エディタプロトコルを用いて実装することも可能です。

- MVCの役割をひとつのオブジェクトに兼務させることもできなくはありませんが、一般には、役割を明確に分割する戦略の方が優れています。こうすることにより、オブジェクトの再利用性や、オブジェクトが使われているプログラムの拡張性が高まります。MVCの役割をひとつのクラスに兼務させる場合も、当該クラスの中心的な役割を決めておくべきです。同じ実装ファイル（保守性を高めるため）カテゴリ分けすることにより、クラスを拡張してそれ以外の役割も担えるようにしてください。
- MVCアプリケーションの設計においては、（少なくとも理論的には）再利用が可能なオブジェクトを、できるだけ多くすることを目標としてください。特にビューオブジェクトやモデルオブジェクトは、再利用性が高いものでなければなりません（既製の仲介型コントローラオブジェクトは、もちろん再利用可能です）。アプリケーション固有の振る舞いは、できるだけコントローラオブジェクトに集約するとよいでしょう。
- ビューがモデルを直接監視して状態変化を検出することも可能ではありますが、できるだけそのような構成にはしないでください。ビューオブジェクトは、常に仲介型コントローラを経由して、モデルオブジェクトの状態変化を知るようにする必要があります。その理由は2つあります。
  - バインディング機構を使って、ビューオブジェクトが直接、モデルオブジェクトのプロパティを監視する構成にすると、NSControllerやそのサブクラスの長所たる機能（選択やプレースホルダの管理、変更を確定/破棄する機能など）を活用できません。
  - バインディング機構を使わない場合、既存のビュークラスのサブクラスを作って、モデルオブジェクトがポストする変更通知を監視できるようにする必要があります。
- アプリケーションを構成するクラス間のコード依存性をできるだけ制限してください。ほかのクラスに対する依存性が大きいほど、再利用性は損なわれます。具体的なガイドラインは、関与する2つのクラスの、MVCパターンにおける役割によって異なります。
  - ビュークラスは、モデルクラスに依存するべきではありません（もっとも、一部のカスタムビューではこれが避けられないこともあります）。
  - ビュークラスは、仲介型コントローラクラスに依存するべきではありません。
  - モデルクラスは、ほかのモデルクラス以外に依存するべきではありません。
  - 仲介型コントローラクラスは、モデルクラスに依存するべきではありません（ただし、ビューと同様、カスタムコントローラクラスではこれが必要な場合もあります）。
  - 仲介型コントローラクラスは、ビュークラスや調整型コントローラクラスに依存するべきではありません。
  - 調整型コントローラクラスは、MVCにおけるあらゆる役割のクラスに依存します。



- プログラミング上のある問題を解決するアーキテクチャをCocoaが提供しており、これがMVCの役割をある特定の種類のオブジェクトに割り当てているならば、そのアーキテクチャに従ってください。こうすればプロジェクトの編成がはるかに容易になります。たとえばドキュメントアーキテクチャには、`NSDocument`オブジェクト（`nib`ファイルにアーカイブされるモデルコントローラ）をファイルの所有者として設定する、Xcodeプロジェクトのテンプレートがあります。

## Cocoa (Mac OS X) におけるModel-View-Controller

Model-View-Controllerデザインパターンは、Cocoaのさまざまな機構や技術の基盤となっています。したがって、オブジェクト指向設計におけるMVCの重要性は、アプリケーションの再利用性や拡張性を高めることだけにとどまりません。アプリケーション自身もMVCパターンに従っていれば、MVCベースであるCocoaの技術を組み込んだとき、その効果をより大きく発揮できることになります。MVCに従って適切に役割を分割していれば、こういった技術を取り入れることに特段の困難はないはずです。逆に分割がうまくいっていなければ、かなりの作業を強いられることになるでしょう。

Mac OS XのCocoaには、Model-View-Controllerを基盤とする、次のようなアーキテクチャ、機構、技術が組み込まれています。

- **ドキュメントアーキテクチャ。**このアーキテクチャのもとでは、ドキュメントベースのアプリケーションは、全体を制御するコントローラオブジェクト（`NSDocumentController`）、各ドキュメントウインドウのコントローラオブジェクト（`NSWindowController`）、各ドキュメントについてコントローラとモデルの役割を兼務するオブジェクト（`NSDocument`）から成ります。
- **バインディング。**MVCはCocoaにおけるバインディング技術の中核です。抽象クラス`NSController`の具象サブクラスとして既製のコントローラオブジェクトがあり、ビューオブジェクトと、適切に設計されたモデルオブジェクトとの間に、バインディングを確立するよう設定できます。
- **アプリケーションのスクリプト制御方式。**アプリケーションをスクリプトで制御できるようにするためには、MVCデザインパターンに従うだけでなく、モデルオブジェクトも適切に設計する必要があります。スクリプトコマンドのうち、アプリケーションの状態にアクセスしたり、ある動作を要求したりするものは、一般に、モデルオブジェクトまたはコントローラオブジェクトに送信されます。
- **Core Data。**Core Dataフレームワークは、モデルオブジェクト間のグラフ（構造）を管理し、これを永続ストアに格納（およびここから検索）することにより、オブジェクトの永続性を保証します。Core DataはCocoaバインディング技術に、緊密に統合されています。MVCもオブジェクトモデリングデザインパターンも、Core Dataアーキテクチャを決定づける要素です。
- **Undo。**Undoアーキテクチャでもモデルオブジェクトは中心的な役割を担います。多くの場合、モデルオブジェクトのプリミティブメソッド（通常はアクセサメソッド）に、取り消し/再実行の処理を実装します。アクションのビュー/コントローラオブジェクトもこの処理に関与します。た

例えば、当該オブジェクトに、取り消し/再実行の機能を起動するメニュー項目に表示する文字列を管理させる、あるいはテキストビューで選択した内容を取り消せるようにする、などが考えられます。

# オブジェクトモデリング

この章では、オブジェクトモデリングやキー値コーディング(key-value coding)に関する用語を説明し、例を示します。いずれもCocoaバインディングやCore Dataフレームワークに特有の概念です。「キーパス(key path)」などの用語を理解することが、この技術を効果的に使うための鍵となります。オブジェクト指向設計やキー値コーディングが初めてという方は、ぜひお読みください。

Core Dataフレームワークを利用するためには、モデルオブジェクトを、ビューやコントローラに依存しない形で記述する手段が必要です。再利用性に優れた設計では、ビューやコントローラがモデルのプロパティにアクセスし、しかも相互に依存関係が生じないような手段がなければなりません。Core Dataフレームワークはこの課題に対処するため、データベース技術、特に「エンティティ-リレーション（実体-関連、entity-relationship）」モデルの考え方や用語を借用しています。

「エンティティ-リレーション」モデルはオブジェクトの表現方法のひとつで、データソースのデータ構造を、オブジェクト指向システムのオブジェクトに対応づけて記述するためによく使われます。これはCocoa独自のものではありません。データベースの分野では広く普及しており、その規則や用語もさまざまな文献に載っています。オブジェクトをデータソースに格納し、ここから検索する処理の記述にも便利な表現方法です。実際のデータソースは、データベース、ファイル、ウェブサービスなど、永続ストアであれば何であっても構いません。データソースの種類には依存しないので、あらゆる種類のオブジェクト、およびほかのオブジェクトとの関係を表現するために使えます。

「エンティティ-リレーション」モデルでは、データを保持するオブジェクトをエンティティ(entities)、その構成要素をアトリビュート(attributes)、データを保持するほかのオブジェクトへの参照をリレーション(relations)と呼びます。アトリビュートとリレーションを併せた概念がプロパティ(properties)です。この3つの単純な構成要素、すなわち、エンティティ、アトリビュート、リレーションを組み合わせ、いくらでも複雑なシステムをモデル化できます。

Cocoaでは、従来型の「エンティティ-リレーション」モデルの規則を若干修正して使い、ここではオブジェクトモデリングと呼びます。オブジェクトモデリングは特に、Model-View-Controller (MVC) デザインパターンでモデルオブジェクトを表現する場合に有用です。これは驚くようなことではありません。どんなに単純なCocoaアプリケーションであっても、モデルは一般に永続的である、すなわち、ファイルなどの「データ収容庫」に格納されるからです。



## エンティティ

エンティティとはモデルオブジェクトのことです。MVCデザインパターンで言うモデルオブジェクトとは、アプリケーションを構成するオブジェクトのうち、所定のデータをカプセル化し、当該データを操作する手段を提供するもののことです。データは一般に永続的ですが、それよりも重要なのは、モデルオブジェクトがデータの表示方法に依存しないことです。

たとえば、モデルオブジェクトを構造化したコレクション（オブジェクトモデル）の形で、企業の顧客ベース、書籍を集めた図書館、コンピュータを集めたネットワークなどを表現できます。図書館の本にはそれぞれ、書名、ISBN番号、著作権表示の日付などのアトリビュートや、著者、図書館利用者などほかのオブジェクトとのリレーションが設定されます。理論上、構成する各部分を識別できる体系は、オブジェクトモデルとして表現できます。

図 8-1に、例として、従業員管理アプリケーションのオブジェクトモデルを示します。このモデルでは、Departmentが部署、Employeeが従業員を表します。

図 8-1 従業員管理アプリケーションのオブジェクト図

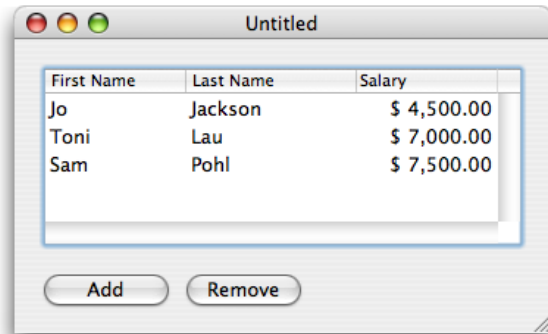


## アトリビュート

アトリビュートは、エンティティの構造のうち、データを含むものを表します。オブジェクトのアトリビュートとしては、スカラ（integer、float、doubleの値）などの単純な値もありますが、Cの構造体（char値の配列、NSPoint構造体など）や、プリミティブクラス（CocoaのNSNumber、NSData、NSColorなど）のインスタンスであることもあります。NSColorのような不変オブジェクトも、通常はアトリビュートと見なします（なお、Core Dataが本来サポートするのは、『*NSAttributeDescription Class Reference*』に示したように、ごく限られたアトリビュート型だけです。しかし、『*Core Data Programming Guide*』の“Non-Standard Persistent Attributes”に示すように、標準以外のアトリビュート型を追加することも可能です）。

Cocoaでは、アトリビュートは一般に、モデルのインスタンス変数やアクセサメソッドに対応します。たとえばEmployeeには、`firstName`、`lastName`、`salary`などのインスタンス変数があります。従業員管理アプリケーションには、Employeeオブジェクトおよびそのアトリビュートのいくつかを表示する、テーブルビューを実装するかも知れません（図 8-2を参照）。表の各行はEmployeeのインスタンスに、各列はEmployeeのアトリビュートに対応します。

図 8-2 従業員のテーブルビュー



First Name	Last Name	Salary
Jo	Jackson	\$ 4,500.00
Toni	Lau	\$ 7,000.00
Sam	Pohl	\$ 7,500.00

## リレーション

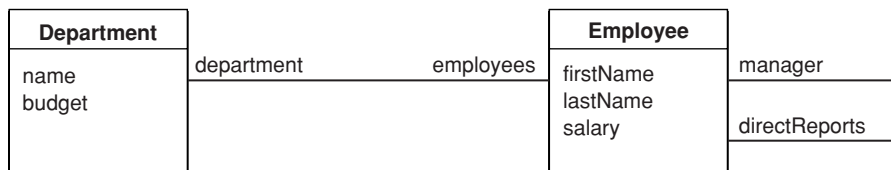
モデルのプロパティにはアトリビュートのほかに、ほかのオブジェクトとのリレーションを表すものもあります。アプリケーションは一般に、いくつかのクラスを組み合わせることでモデルを表します。実行時に現れるオブジェクトモデルは、関連するオブジェクトの集まりであって、これがオブジェクトグラフを構成します。これは一般に永続的オブジェクトです。すなわち、ユーザはこれを生成した後、消えてしまわないよう、アプリケーションを終了する前に、何らかのデータコンテナやファイルに保存します（ドキュメントベースのアプリケーションはその典型的な例）。モデルオブジェクト間のリレーションを実行時にたどることにより、関連するオブジェクトのプロパティにもアクセスできます。

たとえば従業員管理アプリケーションでは、従業員と所属する部署とのリレーション、従業員とその上長とのリレーションがあります。上長も従業員なので、「従業員-上長」のリレーションは反射的リレーション(reflexive relationship)、すなわち、あるエンティティから同じエンティティへのリレーションになっています。

リレーションはその性質上、両方向であるため、少なくとも概念的には、部署とそこに属する従業員とのリレーション、従業員とその部下とのリレーションもあります。図 8-3（59 ページ）に、DepartmentとEmployeeのリレーション、およびEmployeeの反射的リレーションを示します。この例で、エンティティDepartmentのリレーション「employees」は、エンティティEmployeeのリレーション「department」の逆に当たります。しかし、リレーションを一方方向にしかたどれない、すなわち、逆方向のリレーションはない、ということもありえます。たとえば、ある部署にどの従業員が属して

いるか、を調べる必要がないのであれば、このリレーションはモデル化しなくても構いません（一般論としてはそうなのですが、Core Dataでは、Cocoaの一般的なオブジェクトモデリングに制約を追加して、原則として逆方向のリレーションもモデル化することとしています）。

図 8-3 従業員管理アプリケーションに現れる「リレーション」



## リレーションの基数(cardinality)と所有関係(Ownership)

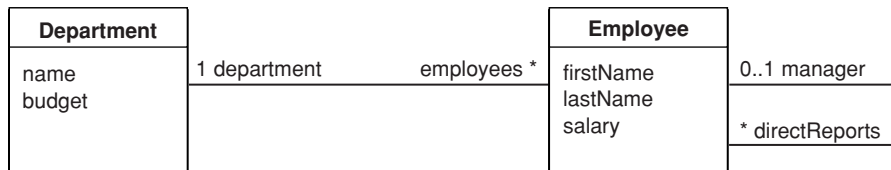
リレーションには**基数**を設定できます。これは、リレーションで結ばれる先に（少なくとも潜在的に）いくつのオブジェクトがあるか、を表します。関係先オブジェクトが常に1つであれば、これを**対1リレーション(to-one relationship)**と呼びます。一方、関係先オブジェクトが複数であっても構わない場合は、**対多リレーション(to-many relationship)**と言います。

リレーションには、必須(mandatory)か任意(optional)かという区別もあります。必須のリレーションとは、関係先を必ず設定しなければならないリレーションのことです。たとえば従業員は、全員どこかの部署に属しなければなりません。一方、任意のリレーションとは、文字通り、関係先がなくてもよいリレーションのことです。たとえば、直属の部下がいない従業員もいます。したがって、[図 8-4](#)（59 ページ）に示したdirectReportsは、任意のリレーションということになります。

基数の範囲を指定することも可能です。任意の対1リレーションは、基数の範囲が0～1ということになります。また、ある従業員の部下の人数は、どのような値にもなりえますが、0～15というように下限と上限を設定することも考えられます（任意の対多リレーションの例）。

図8-4に、従業員管理アプリケーションに現れるリレーションの基数を示します。EmployeeとDepartmentの間には必須の対1リレーションがあります。従業員はいずれか1つの部署に属し、複数の部署に属することはありません。DepartmentとEmployeeの間には任意の対多リレーションがあります（「\*」で表示）。従業員と上長との関係は任意の対1リレーションです（「0-1」で表示）。最上位の従業員に上長はいません。

図 8-4 リレーションの基数



リレーションで結ばれた先のオブジェクトとの間に所有関係がある場合と、そうでない（ほかのオブジェクトと共有する）場合があることにも注意してください。

## プロパティへのアクセス

モデル、ビュー、コントローラがそれぞれ独立していても動作するためには、モデルの実装に依存しない方法で、プロパティにアクセスできるようにする必要があります。これを達成するために、キーと値の組を考えます。

### キー

モデルのプロパティは簡単なキーを使って指定します。多くの場合、これは文字列です。ビューやコントローラは、キーを使って該当するアトリビュートの値を検索します。このように設計すれば、アトリビュートそのものにデータが収容されていなくても構わない、すなわち、間接的に値を取得または導出してもよいことになります。

キー値コーディング（**key-value coding**）とは、この検索を行う機構のことです。オブジェクトのプロパティに間接的に、そして状況によっては自動的に、アクセスすることができます。キー値コーディングでは、オブジェクトのプロパティ（一般にインスタンス変数またはアクセサメソッド）の名前をキーとして使い、その値にアクセスします。

たとえば**Department**オブジェクトの名前は、`name`というキーで取得します。**Department**オブジェクトに`name`というインスタンス変数またはメソッドがあれば、キーに対応する値が返されます（どちらもなければエラー）。同様に**Employee**のアトリビュートも、`firstName`、`lastName`、`salary`などといったキーを使って取得できます。

### 値

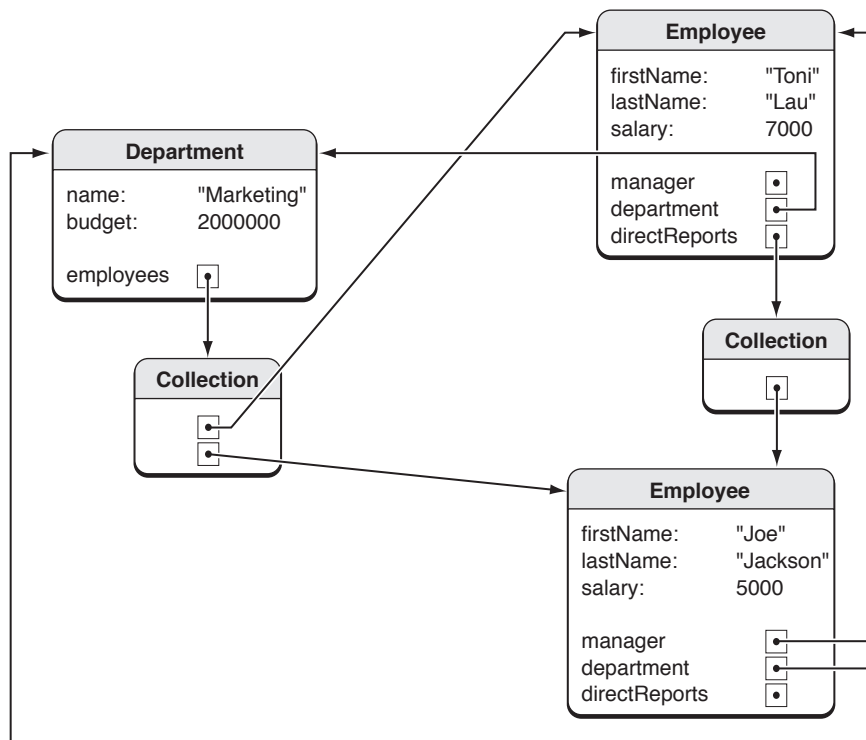
アトリビュートをひとつ固定すると、その値はどのエンティティでもすべて同じデータ型です。アトリビュートのデータ型は、対応するインスタンス変数の宣言で、またはアクセサメソッドの戻り値の型として指定します。たとえば、**Department**オブジェクトの`name`アトリビュートのデータ型は、**Objective-C**の**NSString**オブジェクトである、というようになります。

キー値コーディングにより返されるのは、常にオブジェクト値であることに注意してください。アクセサメソッドの戻り値型やインスタンス変数の型がオブジェクトでない場合は、**NSNumber**や**NSValue**のオブジェクトを生成し、これに値を設定して返すようになっています。たとえば、**Department**の`name`アトリビュートが**NSString**型であれば、キー値コーディングを適用したとき、**Department**オブジェクトの`name`キーに対して返されるのは**NSString**オブジェクトです。一方、**Department**の`budget`アトリビュートが**float**型であれば、**Department**オブジェクトの`budget`キーに対して返されるのは**NSNumber**オブジェクトです。

キー値コーディングを使って値を設定する場合も同様に、対応するアクセサメソッドやインスタンス変数によって決まるデータ型がオブジェクトでなければ、渡されたオブジェクトから「`-typeValue`」メソッドを使って値を抽出し、これを設定するようになっています。

対1リレーションの値は、関係先のオブジェクトです。たとえば、`Employee`オブジェクトの`department`プロパティの値は`Department`オブジェクトです。対多リレーションの値はコレクションオブジェクトです。コレクションの実装方法には、集合と配列の2通りがあります。Core Dataを使っている場合は集合、そうでなければ一般に配列で表され、いずれの場合も関係先のオブジェクトを集めたものになります。たとえば、`Department`オブジェクトの`employees`プロパティの値は、当該部署に属する`Employee`オブジェクトのコレクションです。図 8-5に、従業員管理アプリケーションのオブジェクトグラフの例を示します。

図 8-5 従業員管理アプリケーションのオブジェクトグラフ

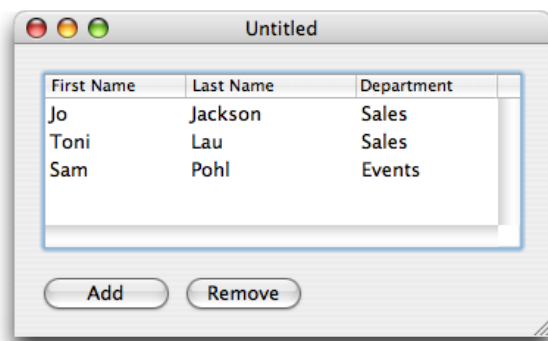


## キーパス

キーパスはドット区切りでキーを連結した文字列で、オブジェクトのプロパティをこの順序でたどる旨を表します。第1のキーのプロパティは関係先のオブジェクトによって決まり、それ以降のキーは直前のプロパティに関係するものとして評価されます。キーパスを使えば、モデルの実装とは独立に、関連するオブジェクトのプロパティを指定できます。オブジェクトグラフをたどって関連するオブジェクトのアトリビュートに到るパスを、いくらでも深くまで指定できます。

キー値コーディングの機構は、与えられたキーパスの値を、キーと値の組と同じようにして検索する処理を実装したものです。たとえば従業員管理アプリケーションで、`Employee`オブジェクトが属する `Department` の名前は、`department.name` というキーパスを使ってアクセスできます。ここで、`department` は `Employee` のリレーション、`name` は `Department` のアトリビュートです。関係先エンティティのアトリビュートを表示するような場合に、キーパスは有用です。たとえば、図 8-6 のような従業員のテーブルビューには、従業員が属する部署オブジェクトの名前が表示されます。部署オブジェクトそのものではありません。「`Department`」列の値には、Cocoa バインディングを使って、表示される配列の要素である `Employee` オブジェクトの、`department.name` がバインドされています。

図 8-6 従業員のテーブルビューに部署名が表示されている様子



キーパスに現れるリレーションすべてに値があるとは限りません。たとえば、ある従業員が CEO であれば、その `manager` リレーションは `nil` となります。このような場合でもキー値コーディングの機構は破綻しません。パスをたどるのをやめて、`nil` などの適切な値を返すだけです。

# オブジェクトの可変性

Cocoaオブジェクトには可変と不変の2種類があります。不変オブジェクトにカプセル化された値は変更できません。いったん生成されると、それが表す値は生涯を通して同じままです。これに対し、可変オブジェクトにカプセル化された値は、いつでも変更できます。以下の各節では、オブジェクトに可変と不変の2種類がある理由や、オブジェクトの可変性に伴う特性や副作用について解説し、可変性が問題になる場合にどう扱えばよいかを説明します。

## オブジェクトに可変と不変の2種類がある理由

特別なことをせずに生成したオブジェクトは可変です。ほとんどのオブジェクトは、「セッタ」と呼ばれるアクセサメソッドを使って、カプセル化されているデータを変更できます。たとえばNSWindowオブジェクトの場合、大きさ、位置、タイトル、バッファの取り扱いなどの特性を変更できます。適切に設計したモデルオブジェクト（たとえば顧客記録を表すオブジェクト）には、インスタンスデータを変更するための「セッタ」メソッドが必須です。

Foundationフレームワークでは、微妙な差異を表すため、可変と不変の2種類のクラスを用意しています。可変クラスは一般に、不変クラスのサブクラスとなっており、クラス名に「Mutable」という文字列が含まれています。その例をいくつか示します。

```
NSMutableArray  
NSMutableDictionary  
NSMutableSet  
NSMutableIndexSet  
NSMutableCharacterSet  
NSMutableData  
NSMutableString  
NSMutableAttributedString  
NSMutableURLRequest
```



---

**注意** AppKitフレームワークのNSMutableParagraphStyleを除き、Foundationフレームワークでは最初から、各クラスに対応して、系統的に名付けた可変クラスが定義されています。これに対しCocoaフレームワークでは、必要ならば可変クラスと不変クラスを用意できるようになっています。

---

可変クラスの名前は変則的ですが、対応する不変クラスに比べ、（mutableという）細かい条件を課しているとも言えます。なぜこのように複雑な仕組みが必要なのでしょう。可変オブジェクトとは別に不変オブジェクトを用意する目的は何でしょうか。

オブジェクトがすべて可変である、という状況を考えてみましょう。アプリケーション内であるメソッドを起動し、文字列を表すオブジェクトの参照が返されたとします。画面上にこの文字列をラベルとして表示しました。この状態で、別のサブシステムが同じ文字列の参照を受け取り、内容を書き換えたとしましょう。すると突然、ラベルの表示が変わってしまいます。さらに悪い事態も考えられます。たとえば、テーブルビューに並べて表示する、配列の参照を受け取ったとします。配列のあるオブジェクトに対応する行をユーザが選択したところ、プログラムのどこか別のところで当該オブジェクトが削除されてしまっていた場合、問題が起こります。不変性とは、いかなる場合でも、知らない間に値が変わってしまうことはない、という保証のことです。

不変性を持たせるのに適した候補としては、個別値のコレクションをカプセル化したオブジェクト、バッファ（文字またはバイト値のコレクション）に格納された値を収容するオブジェクトなどがあります。しかし、このように値をカプセル化するオブジェクトでも、対応する可変クラスを作ることに、常に利益があるとは限りません。単一の単純な値を持つ、NSNumberやNSDateといったオブジェクトは、可変にする候補としてあまり向いていないのです。むしろ、値が変わったことを表すためには、新しい値のインスタンスを別に生成し、置き換えてしまうべきでしょう。

処理性能も、文字列や辞書を表すオブジェクトに不変の版を用意するひとつの理由です。このような基本的なエンティティを可変にすると、若干のオーバーヘッドが生じるからです。記憶域を動的に管理しなければならない（必要に応じて、ある程度まとまったメモリ領域を割り当てたり解除したりする）ので、不変の版よりも処理効率は劣ります。

理論的には、不変性はオブジェクトの値が安定していることを保証しますが、実際には、この保証を常に期待できるとは限りません。不変オブジェクトを返すはずのメソッドが、実際には可変オブジェクトを返すこともあります。その後、この値が変わってしまうと、受け取り手はそれ以前の値のまま変わっていないと想定しているので、処理が破綻してしまいます。オブジェクトの可変性そのものも、さまざまな変換処理を経て変わってしまうことがあります。たとえば、プロパティリストを（NSPropertyListSerializationクラスで）直列化した場合、オブジェクトの可変性は保存されません。元が辞書であったか、配列であったか、という情報が保存されるだけです。したがって、このプロパティリストを元に戻しても、元のオブジェクトとまったく同じにはなりません。元はNSMutableDictionaryオブジェクトであったものが、NSDictionaryオブジェクトに変わってしまいます。



## 可変オブジェクトに関わるプログラミング

オブジェクトの可変性が問題になる場合は、防衛的プログラミングを取り入れるべきでしょう。一般的な規約やガイドラインをいくつか示します。

- 可変オブジェクトを使うのは、生成後、その内容を頻繁に変える必要がある場合のみにしてください。
- 値が変わったとき、新しい値の不変オブジェクトを別に生成し、置き換える方がよい場合もあります。たとえば文字列値をリテインするインスタンス変数には、不変オブジェクトであるNSStringを代入し、セッターメソッドで置き換えるようにするのです。
- 戻り値の型を、不変性を判断する標識として使ってください。
- あるオブジェクトが可変であるか、あるいは可変であるべきか、疑わしい場合は不変にしておいてください。

この節では、ガイドラインの適用が曖昧なところを中心に、可変オブジェクトが関与するプログラミングにおいて選択を求められる典型的な状況について説明します。また、Foundationフレームワークのメソッドのうち、可変オブジェクトを生成するもの、可変か不変かを変換するものについて、概要を説明します。

### 可変オブジェクトの生成と変換

可変オブジェクトは、次の例のように、allocとinitが入れ子になった、標準的なメッセージで生成できます。

```
NSMutableDictionary *mutDict = [[NSMutableDictionary alloc] init];
```

しかし多くの可変クラスは、イニシャライザやファクトリメソッドを提供しています。たとえばNSMutableArrayクラスのinitWithCapacity:メソッドは、オブジェクトの初期（予測）容量を指定できます。

```
NSMutableArray *mutArray = [NSMutableArray arrayWithCapacity:[timeZones count]];
```

容量のヒントを使って、可変オブジェクトのデータ用ストレージを、より効率的に管理できるようになっているのです（クラスファクトリメソッドは自動解放インスタンスを返す規約になっているので、コード中で生存状態に保ちたければ、オブジェクトを「リテイン(retain、保持)」しなければなりません）。

また、汎用型の既存のオブジェクトから可変のコピーを作ることにより、可変オブジェクトを生成することもできます。これは、Foundationの可変クラスのスーパークラス（不変版）に実装されている、mutableCopyメソッドで行います。

```
NSMutableSet *mutSet = [aSet mutableCopy];
```

逆に、可変オブジェクトにcopyメッセージを送ることにより、その不変なコピーを作ることができます。

可変と不変の2種類があるFoundationクラスの多くに、次のような相互変換メソッドが実装されています。

- `typeWithType` : -たとえば`arrayWithArray` :
- `setType` : -たとえば`setString` : (可変クラスのみ)
- `initWithType:copyItems` : -たとえば`initWithDictionary:copyItems` :

## 可変インスタンス変数を保存する、または戻り値として返す場合の取り扱い

Cocoa開発では、インスタンス変数を可変にするか不変にするか、という判断が頻繁に発生します。辞書、文字列など、値が変化するインスタンス変数について、これを可変にするのが適切なもののような場合でしょうか。逆に、オブジェクトを不変にし、値が変わった場合には別にオブジェクトを生成して置き換える、という方式が適切なもののような場合でしょうか。

一般に、内容全体が一括して変化する場合、不変オブジェクトが適しています。文字列 (NSString) やデータオブジェクト (NSData) は、通常これに当てはまります。データが徐々に追加される形で変化する場合は、可変にする方が理に適っています。配列、辞書などのコレクションはこれに当てはまります。しかし、変化の頻度やコレクションの大きさも考慮して判断しなければなりません。たとえば、ほとんど変化することのない小さな配列であれば、不変にする方がよいでしょう。

インスタンス変数としてリテインするコレクションの可変性を判断する際には、次の2点も考慮してください。

- 頻繁に変化する可変コレクションがあって、これをクライアントに頻繁に渡す (すなわち、「ゲッタ」アクセサメソッドで直接返す) 場合、クライアントにとっては、参照しているものが知らない間に変わってしまう危険があります。これが実際に発生しうるものであれば、インスタンス変数は不変にしなければなりません。
- インスタンス変数の値が頻繁に変わるとしても、ゲッタメソッドでクライアントに返すことが稀であれば、インスタンス変数自体は可変にしておき、アクセサメソッドでその不変なコピーを生成して返す、という方法もあります。メモリマネージドプログラムでは、このオブジェクトは自動解放です (リスト 9-1を参照) 。

#### リスト 9-1 可変インスタンス変数の不変なコピーを返すコード例

```
@interface MyClass : NSObject {  
    // ...  
    NSMutableSet *widgets;  
}  
// ...  
@end  
  
@implementation MyClass  
- (NSSet *)widgets {  
    return (NSSet *)[[widgets copy] autorelease];  
}
```

可変コレクションを戻り値としてクライアントに返すための巧妙な方法として、オブジェクトが可変か不変かを表すフラグを管理する、というものがあります。変化が起こった場合、オブジェクトを可変にしたうえで変更を施します。コレクションを戻り値として返す際には、（必要ならば）不変にした上で返します。

## 可変オブジェクトの受け取り

メソッドを起動した側では、次の2つの理由で、戻り値オブジェクトの可変性を意識しなければなりません。

- オブジェクトの値を変更できるかどうか知るため。
- 参照中、知らない間に値が変化している可能性がないか確認するため。

## イントロスペクションではなく戻り値型を使うこと

戻り値オブジェクトを変更できるかどうかの判断は、戻り値の正式な型に基づいて行うべきです。たとえば、不変と型づけられた配列オブジェクトを受け取った場合、変更しようとしてはなりません。属するクラスに基づいて可変か否かを判断する方法は、プログラムの作法として許容できません（次の例を参照）。

```
if ( [anArray isKindOfClass:[NSMutableArray class]] ) {  
    // anArrayにオブジェクトを追加、あるいはanArrayからオブジェクトを削除  
}
```

実装上の理由で、この場合、`isKindOfClass:`の戻り値が適切であるとは限りません。そうでなくても、属するクラスに基づいてオブジェクトが可変か否か判断できる、という想定をするべきではありません。当該オブジェクトを返すメソッドのシグニチャが、可変性についてどのように宣言しているか、のみをもとに判断するべきです。はっきりしない場合は不変であると想定してください。

このガイドラインが重要である理由は、次の2つの例を見るとはっきりするでしょう。

- プロパティリストをファイルから読み込んでいます。**Foundation**フレームワークは、リストを順次読み込み、値が等しい要素についてはオブジェクトを別々に生成せず共有する形で、プロパティリストオブジェクトを構築しました。その後、このプロパティリストオブジェクトを参照し、要素のいくつかを変更しました。するといつの間にか、変更していないはずの要素も値が変化していました。
- `NSView`に対して (`subviews`メソッドで) サブビューを要求し、戻り値として`NSArray`と宣言されたオブジェクトを取得しましたが、内部的には、これは`NSMutableArray`でした。この配列を別のコードに渡し、イントロスペクションによりこれは可変であると判断したので、実際に変更しました。しかし、この変更により、`NSView`クラス内部のデータ構造を変えてしまったことになります。

オブジェクトの可変性を、イントロスペクションの結果に基づいて判断しないでください。可変か否かは、APIを呼び出すコードの静的な意味（すなわち、戻り値型）に基づいて判断するべきです。クライアントに渡す際、可変か否かを曖昧さなく伝える必要がある場合は、オブジェクトにフラグの形でその情報を付加してください。

## 受け取ったオブジェクトのスナップショット作成

メソッドから受け取ったオブジェクトが不変であり、知らない間に変化することがないことを保証したい場合は、ローカルにコピーしたスナップショットを作っておくとよいでしょう。必要の都度、保存しておいたオブジェクトと現在の状態とを比較するのです。変化していることが分かった場合は、プログラム中、不変であるという前提に依存している箇所を修正してください。リスト 9-2にこの技法の実装例を示します。

### リスト 9-2 書き換えられる可能性があるオブジェクトのスナップショット作成

```
static NSArray *snapshot = nil;

- (void)myFunction {
    NSArray *thingArray = [otherObj things];
    if (snapshot) {
        if ( ![thingArray isEqualToArray:snapshot] ) {
            [self updateStateWith:thingArray];
        }
    }
}
```

```
    }  
    }  
    snapshot = [thingArray copy];  
}
```

スナップショットを作成し後で比較する方式には、高負荷になるという問題があります。同じオブジェクトのコピーを複数作成することになるからです。より効率のよい代替案として、キー値監視を用いる方法があります。このプロトコルについては『*Key-Value Observing Programming Guide*』を参照してください。

## コレクション中の可変オブジェクト

可変オブジェクトをコレクションに格納すると、問題が起こる場合があります。コレクションによっては、収容するオブジェクトが変化するとコレクション内の配置に影響するため、無効になったり、場合によっては破損したりする恐れがあります。まず、オブジェクトのあるプロパティが、`NSDictionary`や`NSSet`のようなハッシュ方式のコレクションのキーになっているとしましょう。すると、このプロパティが変化した結果、`hash`メソッドや`isEqual:`メソッドの戻り値が変わった場合、コレクションそれ自体が破損してしまいます（コレクションに収容したオブジェクトの`hash`メソッドが、その内部状態に依存しないのであれば、破損は免れるかも知れません）。第2に、順序つきコレクション（整列配列など）に収容されたオブジェクトのプロパティ値が変化すると、配列中のほかのオブジェクトと比較したときの結果にも影響し、順序が正しくなくなる恐れがあります。

# アウトレット

アウトレットとは、オブジェクトのプロパティのうち、ほかのオブジェクトを参照するもののことです。参照はInterface Builder上で接続することによりアーカイブされます。参照元オブジェクトとそのアウトレットとは、オブジェクトがnibファイルから展開（unarchive）される都度、接続を確立し直されます。オブジェクトは保持するアウトレットを、IBOutletという型修飾子とweakオプションをつけ、プロパティとして宣言します。次に例を示します。

```
@interface AppController : NSObject
{
}

@property (weak) IBOutlet NSArray *keywords;
```

アウトレットはプロパティなので、オブジェクトにカプセル化されたデータの一部を成し、インスタンス変数の形で保持されます。しかしアウトレットは、単なるプロパティではありません。オブジェクトとそのアウトレットの接続は、nibファイルにアーカイブされます。nibファイルを読み込む際に、この接続も展開し、確立し直すようになっています。したがって、ほかのオブジェクトにメッセージを送る必要が生じた時点では、常に接続された状態になっています。型修飾子IBOutletは、プロパティ宣言に適用されるタグです。Interface Builderはこのタグに基づいて、これがアウトレットであると認識し、その表示や接続をXcodeと同期します。

アウトレットを弱い参照（weak）と宣言しているのは、強い参照サイクルを回避するためです。

アウトレットの作成と接続は、Xcodeに付属するInterface Builderで行います。アウトレットのプロパティ宣言には、IBOutlet修飾子でタグづけする必要があります。

アプリケーションは一般に、カスタムコントローラオブジェクトと、ユーザインターフェイス上のオブジェクトとの間に、アウトレット接続を設定します。しかしこの接続は、Interface Builder上でインスタンスとして表せるオブジェクトであればどれとどれの間でも（カスタムオブジェクトどうしでも）可能です。オブジェクトの状態を表すほかの設定項目と同様、アウトレットについても、クラスに含めるのが適切かどうか、きちんと検討するべきです。オブジェクトのアウトレットが多ければ、それだけメモリ消費量も増えるからです。行列のインデックス位置に基づいて検索する、関数引数として指定する、タグ（割り当て済みの数値識別子）を使って検索するなど、オブジェクトの参照を取得するほかの方法があるならばそれを採用してください。

アウトレットはオブジェクト合成の一形態です。オブジェクトが何らかの方法で、構成要素であるオブジェクトの参照を実行時に（動的に）取得し、これにメッセージを送信できるようにする、という形のパターンです。一般に、この参照先オブジェクトは、プロパティ（内部的にはインスタンス変数）として保持します。この変数は、プログラム実行中のある時点で、適切な参照を与えて初期化しなければなりません。

# Receptionistパターン

Receptionistデザインパターンは、アプリケーションのある実行コンテキストで発生したイベントを、別の実行コンテキストにリダイレクトして処理する際に起こる、さまざまな問題に対処するものです。これは混成型のパターンです。「Gang of Four (GoF)」の書籍に出てくる23種類には入っていませんが、ここに説明されているCommand、Memo、Proxyの3つのデザインパターンを要素として組み合わせたものです。（やはりこの本には出ていない）Trampolineパターンの変形と考えることもできます。このパターンでは、最初にイベントを受け取るオブジェクトを「トランポリン」といいます。即座にイベントを「跳ね返し」て、実際に処理するオブジェクトにリダイレクトするからです。

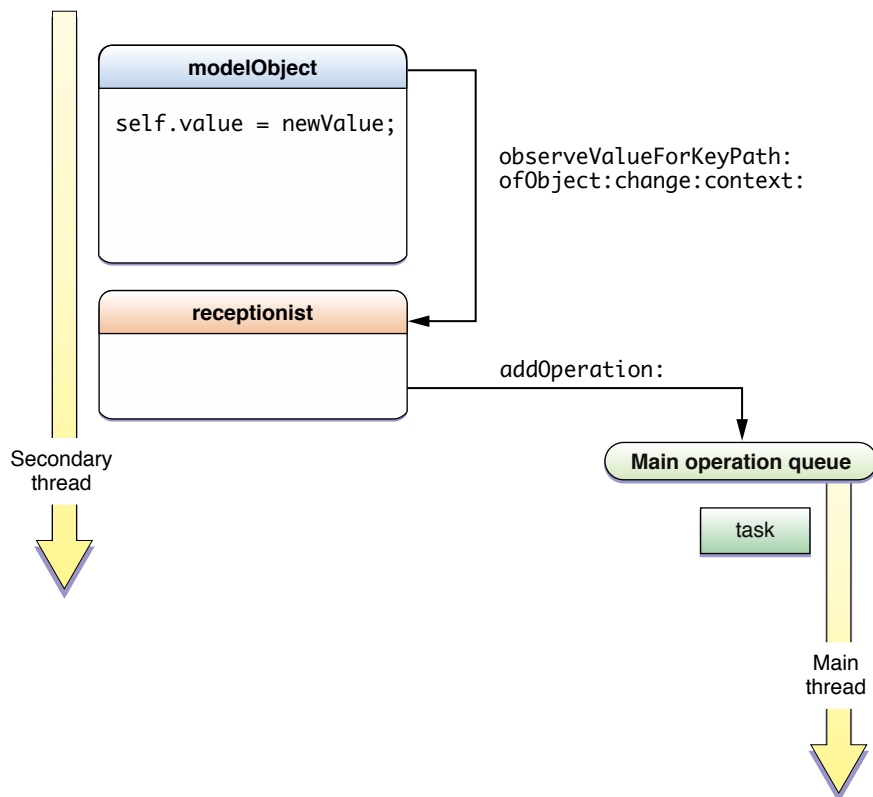
## 実践的なReceptionistデザインパターン

キー値監視（KVO）による通知があると、オブザーバに実装された `observeValueForKeyPath:ofObject:change:context:` メソッドが起動されます。副スレッドでプロパティ値が変化すると、同じスレッド上で `observeValueForKeyPath:ofObject:change:context:` のコードが実行されます。ここで、このパターンの中核オブジェクトであるReceptionist（受付係）が、スレッドの仲介役として動作します。図 11-1 に示すように、Receptionist オブジェクトは、モデルオブジェクトのプロパティのオブザーバとして割り当てられます。Receptionist には `observeValueForKeyPath:ofObject:change:context:` を実装します。これは、副スレッドが受け取った通知を、別の実行コンテキスト（この場合は主操作キュー）にリダイレクトします。プロパ



ティ値が変化すると、ReceptionistにはKVO通知が届きます。Receptionistは即座に、ブロック操作を主操作キューに追加します。ブロックには、クライアントが指定した、ユーザインターフェイスを適切に更新するコードが含まれています。

図 11-1 キー値監視による更新通知を主操作キューに「跳ね返らせる」様子



Receptionistクラスには、自分自身をプロパティのオブザーバとして追加し、KVO通知を更新タスクに変換するために必要な事項を定義します。したがって、どのオブジェクトを監視しているか、オブジェクトのどのプロパティを監視しているか、どの更新タスクを実行するか、どのキュー上で実行するか、を知らなければなりません。リスト 11-1に、RCReceptionistクラスの宣言の先頭部分と、そのインスタンス変数を示します。

リスト 11-1 Receptionistクラスの宣言

```
@interface RCReceptionist : NSObject {
    id observedObject;
    NSString *observedKeyPath;
    RCTaskBlock task;
    NSOperationQueue *queue;
}
```

インスタンス変数`task`は、次のように宣言された型のブロックオブジェクトです。

```
typedef void (^RCTaskBlock)(NSString *keyPath, id object, NSDictionary *change);
```

この引数並びは、`observeValueForKeyPath:ofObject:change:context:`メソッドの引数に似ています。次に、引数クラスにはクラスファクトリメソッドがひとつ宣言されており、そこでは `RCTaskBlock` オブジェクトが引数となっています。

```
+ (id)receptionistForKeyPath:(NSString *)path
    object:(id)obj
    queue:(NSOperationQueue *)queue
    task:(RCTaskBlock)task;
```

このメソッドは、**Receptionist** オブジェクトを生成して、渡された値をインスタンス変数に代入し、このオブジェクトをモデルオブジェクトのプロパティのオブザーバとして追加します（リスト 11-2 を参照）。

#### リスト 11-2 Receptionist オブジェクトを生成するクラスファクトリメソッド

```
+ (id)receptionistForKeyPath:(NSString *)path object:(id)obj queue:(NSOperationQueue *)queue task:(RCTaskBlock)task {
    RCReceptionist *receptionist = [RCReceptionist new];
    receptionist->task = [task copy];
    receptionist->observedKeyPath = [path copy];
    receptionist->observedObject = [obj retain];
    receptionist->queue = [queue retain];
    [obj addObserver:receptionist forKeyPath:path
        options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
        context:0];
    return [receptionist autorelease];
}
```

このコードでは、ブロックオブジェクト（`task`）をリテンションするのではなく、コピーしていることに注意してください。ブロックは恐らくスタック上に生成されているので、KVO通知が配送される時点までメモリ上に存在するよう、ヒープ上にコピーしておかなければなりません。

最後に、引数クラスに `observeValueForKeyPath:ofObject:change:context:` メソッドを実装します。実際の中身はごく単純です（リスト 11-3 を参照）。

## リスト 11-3 KVO通知の処理

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
    change:(NSDictionary *)change context:(void *)context {
    [queue addOperationWithBlock:^(
        task(keyPath, object, change);
    )];
}
```

このコードは、タスクを指定された操作キューに入れているだけです。その際、タスクブロックに、監視されるオブジェクト、変化したプロパティのキーパス、新しい値が入った辞書を渡します。タスクは、キュー上で自分自身を実行する、`NSBlockOperation`オブジェクト内にカプセル化されています。

クライアントオブジェクトは、**Receptionist**オブジェクトを生成する際、ユーザインターフェイスを更新するブロックコードを渡します（リスト 11-4を参照）。このときクライアントは、ブロックを実行する操作キュー（この場合は主操作キュー）も渡していることに注意してください。

## リスト 11-4 Receptionistオブジェクトの生成

```
RCReceptionist *receptionist = [RCReceptionist
receptionistForKeyPath:@"value" object:model queue:mainQueue task:^(NSString
*keyPath, id object, NSDictionary *change) {
    UIView *viewForModel = [modelToViewMap objectForKey:model];
    UIColor *newColor = [change objectForKey:NSKeyValueChangeNewKey];
    [[[viewForModel subviews] objectAtIndex:0] setFillColor:newColor];
}];
```

## Receptionist/パターンが向いている状況

**Receptionist**デザインパターンは、ある処理を別の実行コンテキストに「跳ね返して」処理させる必要がある場合に、いつでも適用できます。通知を監視する、ブロックハンドラを実装する、アクションメッセージに応答する、などの状況で、コードが適切な実行コンテキストで実行されるようにしたい場合、**Receptionist**/パターンを実装することにより、処理を当該実行コンテキストにリダイレクトすることができます。単にそのまま「跳ね返す」だけでなく、データから不要なものを除去したり併合したりした上で、タスクを「跳ね返して」処理させることも可能です。あるいは、収集したデータを集積しておき、ある時間間隔で実行コンテキストにリダイレクトすることも考えられます。

Receptionistパターンが役に立つ状況としては、キー値監視もあります。キー値監視では、モデルオブジェクトのプロパティ値が変化すると、KVO通知の機構を介してオブザーバに通知が届きます。しかし、モデルオブジェクトの変化は、バックグラウンドスレッドで発生することもあります。その結果、スレッド間の不整合が起こります。モデルオブジェクトの状態が変化すれば、一般にユーザーインターフェイスを更新することになりますが、これは主スレッド上でしか実行できないからです。この場合、KVO通知を主スレッドにリダイレクトして、ユーザーインターフェイスを更新させるとよいでしょう。

# ターゲット-アクション機構

デリゲート、バインディング、通知などの機構は、プログラムで、ある種の形式のオブジェクト間通信を処理する方法として有用ですが、外観（表示）に直接関わるような通信に、特に適しているということはありません。典型的なアプリケーションのユーザインターフェイスは、多数の図形オブジェクトから成りますが、中でも多用されているのはコントロール部品でしょう。コントロール部品とは、現実世界にあるデバイス、あるいは論理的なデバイス（ボタン、スライダ、チェックボックスなど）を、図形で模したものです。ラジオのチューナのような、現実世界の制御部品と同様に、これを部品して持つシステム、すなわちアプリケーションに、意図を伝えるために使います。

ユーザインターフェイスにおけるコントロール部品の役割は単純です。ユーザの意図を解釈し、ほかのオブジェクトにその要求を伝えるのです。クリックする、Returnキーを押すなど、ユーザがコントロール部品に何らかの働きかけをすると、ハードウェアデバイスが生のイベントを生成します。コントロール部品はイベントを（Cocoa用に適切にパッケージ化した形で）受け付け、アプリケーションに応じた命令に翻訳します。しかしイベントそれ自身が、ユーザの意図に関する情報を与えるわけではありません。単にマウスボタンをクリックした、あるいはキーを押した旨を表すだけです。したがって、イベントを命令に翻訳する、何らかの機構を呼び出す必要があります。この機構をターゲット-アクションと呼びます。

Cocoaはターゲット-アクション機構を、コントロール部品とほかのオブジェクトとの通信に利用しています。コントロール部品（Mac OS Xの場合はそのセル）はこの機構を利用して、アプリケーションに特有の命令を適切なオブジェクトに送信するために必要な情報をカプセル化するので、受け取り手のオブジェクト（一般にカスタムクラスのインスタンス）をターゲットといいます。アクションとは、コントロール部品がターゲットに送るメッセージのことです。ユーザイベントを受け取ろうとするオブジェクト（すなわちターゲット）は、当該イベントに「意味」を与えるオブジェクトでもあります。この「意味」はアクションの名前に反映されるのが通例です。

## ターゲット

ターゲットはアクションメッセージの受け取り手です。コントロール部品（あるいは多くの場合そのセル）はアクションメッセージのターゲットを、アウトレットの形で保持します（“[アウトレット](#)”（70 ページ）を参照）。ターゲットはカスタムクラスのインスタンスであることが普通ですが、実際にはどのCocoaオブジェクトでも、そのクラスに適切なアクションメソッドを実装すれば、ターゲットにすることができます。

また、セルやコントロール部品のターゲット（アウトレット）を`nil`としておき、実行時に決めることも可能です。ターゲットが`nil`である場合、アプリケーションオブジェクト（`NSApplication`または`UIApplication`）は次の順序で、適切な受け取り手を検索するようになっています。

1. まず、キーウインドウのファーストレスポンドを探します。次に、`nextResponder`リンクを使ってレスポンドチェーンをたどります。このチェーンは最終的に、ウインドウオブジェクト（`NSWindow`または`UIWindow`）のコンテンツビューに達します。

---

**注意** Mac OS Xのキーウインドウは、キーの押下に応答するほか、メニューやダイアログからのメッセージも受け取ります。一方、アプリケーションの主ウインドウは、ユーザアクションが集約される中心的な場所（フォーカス）であり、キーの状態も保持しています。

---

2. ウインドウオブジェクト、次いでそのデリゲートが受け取り手となりうるかどうか試みます。
3. 主ウインドウがキーウインドウと異なる場合は、主ウインドウのファーストレスポンドから順次レスポンドチェーンをたどって、最終的にウインドウオブジェクトおよびそのデリゲートに達するまで検索します。
4. 続いて、アプリケーションオブジェクトが応答できるかどうか試みます。できない場合はそのデリゲートを試みます。アプリケーションオブジェクトおよびそのデリゲートは、受け取り手の候補としては最終です。

コントロール部品はそのターゲットをリテインしません（するべきではありません）。しかし、アクションメッセージを送信するコントロール部品のクライアント（通常はアプリケーション）は、ターゲットが確実にアクションメッセージを受け取れるようにする責任を負います。用心のため、ターゲットをメモリマネージド環境にリテインしなければならないかも知れません。これはデリゲートやデータソースに関しても同様です。

## アクション

アクションとは、コントロール部品がターゲットに送るメッセージのことです。ターゲット側から見れば、アクションメッセージに応答するため、ターゲットに実装するメソッドのことになります。コントロール部品（AppKitの場合は一般にコントロール部品のセル）は、アクションをSEL型（セレクト型）のインスタンス変数として保存しています。SELはObjective-Cのデータ型で、メッセージのシグニチャを指定するために使います。アクションメッセージのシグニチャは単純かつ独特です。起動されるメソッドは、戻り値がなく、通常は`id`型の引数だけを取ります。この引数を`sender`と呼ぶ規約になっています。NSResponderクラスからひとつ例を示しましょう。このクラスにはアクションメソッドがいくつも定義されています。

```
- (void)capitalizeWord:(id)sender;
```

アクションメソッドはほかにもいくつかのCocoaクラスに宣言されていますが、いずれもシグニチャは同じです。

```
- (IBAction) deleteRecord:(id)sender;
```

ここに現れるIBActionは、戻り値型の指定ではなく型修飾子です（したがって戻り値はありません）。アプリケーション開発時、Interface Builderはこの型修飾子を手掛かりとして、プログラムで追加されたアクションを、プロジェクトで定義されるアクションメソッドを内部的に保持するリストに同期させます。

**iosにおける注意事項** UIKitでは、アクションセレクタの形式がほかに2種類あります。詳しくは[“UIKitのターゲット-アクション機構”](#)（83 ページ）を参照してください。

sender引数は通常、アクションメッセージを送信したコントロール部品を表します（実際の送信元以外のオブジェクトに差し替えられている場合もあります）。この考え方は、葉書の返信先住所に似ていると言えるでしょう。ターゲットは、必要ならば、送信元にさらに情報を要求できます。senderが実際の送信元以外のオブジェクトに差し替えられている場合も、当該オブジェクトを同様に扱ってください。たとえば、テキストフィールドにユーザがテキストを入力したとき、ターゲットで起動されるメソッドとして、次のnameEntered:のようなものが考えられます。

```
- (void)nameEntered:(id) sender {
    NSString *name = [sender stringValue];
    if (![name isEqualToString:@""]) {
        NSMutableArray *names = [self nameList];
        [names addObject:name];
        [sender setStringValue:@""];
    }
}
```

この応答メソッドは、テキストフィールドから内容を取り出し、文字列を配列に追加してインスタンス変数としてキャッシュした後、フィールドをクリアしています。送信元に対する問い合わせの内容としてはほかに、NSMatrixオブジェクトに対して選択状態の行を訊ねる（[sender selectedRow]）、NSButtonオブジェクトに対してその状態を訊ねる（[sender state]）、コントロール部品に対応するセルに対してそのタグ（数値識別子）を訊ねる（[[sender cell] tag]）、などが考えられます。

## AppKitフレームワークにおけるターゲット-アクション

AppKitフレームワークでは、「ターゲット-アクション」機構を実装するために、特別なアーキテクチャと規約を設けています。

### コントロール部品、セル、メニュー項目

AppKitのコントロール部品は、多くが`NSControl`クラスから継承しているオブジェクトです。コントロール部品には、アクションメッセージをターゲットに送信する責務がありますが、メッセージを送るために必要な情報を運ぶことはほとんどありません。この目的にはセルを使うのが普通です。

大部分のコントロール部品にはセル（`NSCell`から派生したオブジェクト）が関連づけられています。なぜこのような関連づけが必要なのでしょう。コントロール部品はかなり「重い」オブジェクトです。`NSView`、`NSResponder`など、多くの祖先のインスタンス変数をすべて継承しているからです。そこで、セルを使って画面上の領域をいくつかの機能領域に分割します。セルは軽量オブジェクトで、いくつか集まってコントロール部品の全体または一部を覆います。単に領域を分割するだけでなく、処理も分割しています。すなわち、コントロール部品が実行すべき描画処理の一部を分担し、本来コントロール部品が保持すべきデータの一部を保持しているのです。ここに言うデータとは、ターゲットおよびアクションのインスタンス変数です。[図 12-1](#)（81 ページ）にコントロール部品とセルのアーキテクチャを示します。

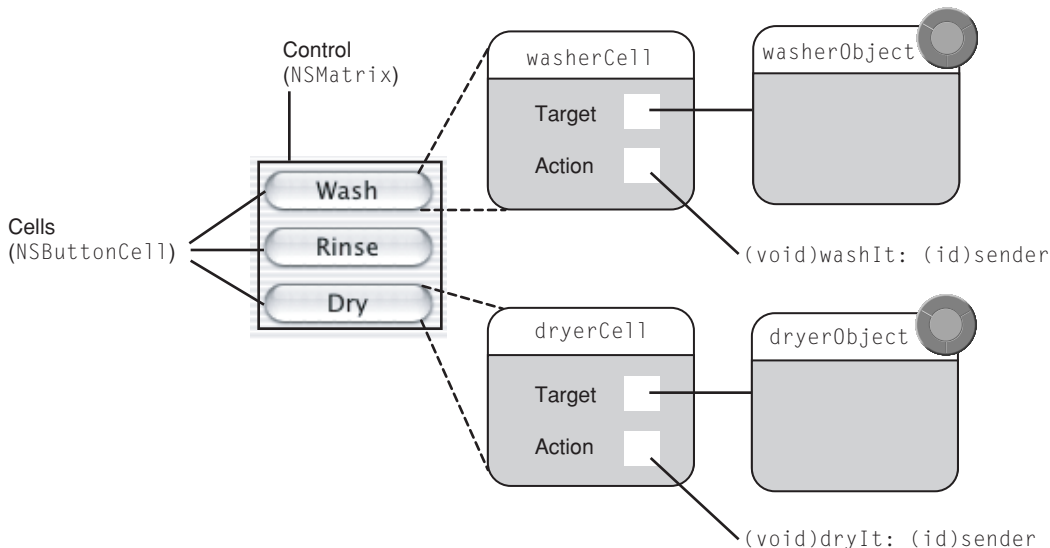
`NSControl`も`NSCell`も抽象クラスなので、ターゲットやアクションのインスタンス変数の設定処理は不完全です。デフォルトでは、`NSControl`は単に、関連するセルの情報があればそれを設定するだけです（`NSControl`自身は、自分とセルを結ぶ、1対1の対応関係しか管理できません。ただし、`NSMatrix`など、`NSControl`のサブクラスの中には、複数のセルとの関係を管理できるものもあります）。`NSCell`のデフォルト実装は、単に例外を投げるだけになっています。継承階層を1段階降りると、ターゲットやアクションの設定処理を実際に実装している、`NSActionCell`クラスがあります。

`NSActionCell`から派生したオブジェクトは、ターゲットやアクションの値を対応するコントロール部品に渡すようになっているので、コントロール部品はアクションメッセージを組み立て、適切なレシーバに送ることができるのです。`NSActionCell`オブジェクトはコントロール部品に代わって、マウス（カーソル）を追跡する処理の一部を担当します。すなわち、該当する領域を強調表示し、アク



ションメッセージを所定のターゲットに送信する処理です。多くの場合NSControlオブジェクトは、対応するNSActionCellオブジェクトに、外観や振る舞いの実現を委ねます（NSMatrixやそのサブクラスであるNSFormは例外で、NSControlのサブクラスですが、この方式には従いません）。

図 12-1 コントロール部品とセルから成るアーキテクチャにおける、ターゲット-アクション機構の動作原理



ユーザがメニューからある項目を選択すると、アクションがターゲットに送信されます。アーキテクチャとして見ると、メニュー（NSMenuオブジェクト）やメニュー項目（NSMenuItemオブジェクト）は、コントロール部品やセルから完全に独立しています。NSMenuItemクラスには「ターゲット-アクション」機構が実装されており、インスタンスはこれを利用できます。NSMenuItemオブジェクトには、ターゲットとアクションを管理するインスタンス変数（および関連するアクセサメソッド）がどちらもあって、ユーザがメニュー項目を選択したとき、アクションメッセージをターゲットに送信するようになっています。

**注意** コントロール部品とセルから成るアーキテクチャについて詳しくは、『*Control and Cell Programming Topics for Cocoa*』および『*Application Menu and Pop-up List Programming Topics*』を参照してください。

## ターゲットやアクションの設定

セルやコントロール部品のターゲットやアクションは、プログラムで、あるいはInterface Builder上で設定できます。通常はInterface Builderで設定するようお勧めします。こうすれば正しく設定されたことがInterface Builder上で視覚的に確認できるほか、接続をロックし、nibファイルにアーカイブすることも可能です。次に示すように、その手順は簡単です。

1. アクションメソッドを、IBAction修飾子を持つカスタムクラスのヘッダファイルに宣言します。

2. **Interface Builder**上で、メッセージの送信元コントロール部品を、ターゲットのアクションメソッドに接続します。

アクションを処理するのが、カスタムクラスのスーパークラス、あるいはAppKitやUIKitの既製のクラスであれば、アクションメソッドを宣言しなくても接続できます。もちろん、宣言した場合は、そのメソッドを実装しなければなりません。

アクションやターゲットをプログラムで設定する場合は、次のメソッドで、コントロール部品やセルオブジェクトにメッセージを送信します。

```
- (void)setTarget:(id)anObject;  
- (void)setAction:(SEL)aSelector;
```

上記のメソッドの使用例を示します。

```
[aCell setTarget:myController];  
[aControl setAction:@selector(deleteRecord:)];  
[aMenuItem setAction:@selector(showGuides:)];
```

ターゲットやアクションをプログラムで設定する方法にも利点はあります。この方法でしか設定できない、という状況もあります。たとえば、実行時の条件（ネットワーク接続の有無、インスペクタウインドウが読み込み済みか否か、など）に応じて、ターゲットやアクションを変えたい、という場合が考えられます。あるいは、ポップアップメニューの項目を動的に入れ替え、その項目ごとにアクションを設定したい場合も同様です。

## AppKitで定義されているアクション

AppKitフレームワークには、アクションメッセージを送信するNSActionCellベースのさまざまなコントロール部品に加え、多くのクラスにアクションメソッドが定義されています。その中には、Cocoaアプリケーションプロジェクトを作成した時点で、デフォルトターゲットに接続されているアクションもあります。たとえば、アプリケーションメニューの「Quit」コマンドは、グローバルアプリケーションオブジェクト（NSApp）のterminate:メソッドに接続されています。

NSResponderクラスにも、テキストを対象とする、よく使われる操作に対応して、数多くのデフォルトアクションメッセージ（標準コマンドともいう）が定義されています。したがって、Cocoaテキストシステムはこのアクションメッセージを、アプリケーションのレスポндаチェーン（イベントを処理するオブジェクトの階層シーケンス）をたどって、対応するメソッドを実装している、NSView、NSWindow、NSApplicationの各オブジェクトに送信できます。

## UIKitのターゲット-アクション機構

UIKitフレームワークにもさまざまなコントロールクラスが宣言、実装されています。このフレームワークのコントロールクラスは、iOS用のターゲット-アクション機構の多くを定義している、`UIControl`クラスを継承しています。しかしAppKitとUIKitでは、ターゲット-アクション機構の実装方法に根本的な違いがあります。そのひとつに、UIKitには本当の意味でのセルクラスがない、という点があります。UIKitのコントロール部品は、ターゲットやアクションに関する情報の処理をセルに委ねません。

もっと重要な違いが、イベントモデルの性質にあります。AppKitフレームワークの場合、ユーザは一般に、マウスやキーボードを操作して、システムが処理すべきイベントを登録します。このイベント（ボタンをクリックするなど）には制約があり、また、個々のイベントは独立しています。したがってAppKitのコントロールオブジェクトは、ひとつひとつの物理イベントが、ターゲットにアクションを送信する「トリガ」である、と認識します（ボタンの場合は「マウスアップ」イベントがトリガ）。一方、iOSの場合、マウスのクリックやドラッグ、あるいは物理的なキーストロークではなく、ユーザの指がイベントを発生させます。画面上のオブジェクトに、同時に2本以上の指が触れ、しかもその指が別々の方向に動くことさえあるのです。

この「マルチタッチ」イベントモデルを考慮して、UIKitでは一連のコントロールイベント定数を`UIControl.h`に宣言しています。ユーザがコントロール部品に対して施しうるさまざまな物理的ジェスチャを表すもので、コントロール部品から指を持ち上げる、コントロール部品内に指をドラッグする、テキストフィールド内でタッチダウンする、などさまざまな動作を表現できます。コントロールオブジェクトは、こういったタッチイベントに応じて、アクションメッセージをターゲットに送信するように設定できます。UIKitのコントロールクラスの多くは、所定のコントロールイベントを生成するように実装されています。たとえばUISliderクラスのインスタンスは`UIControlEventValueChanged`コントロールイベントを生成するので、これを使ってアクションメッセージをターゲットオブジェクトに送信できます。

コントロール部品がアクションメッセージをターゲットオブジェクトに送信するように設定するためには、ターゲットやアクションを、コントロールイベントに対応づけます。これは、指定しようとするターゲットとアクションの組ごとに、`addTarget:action:forControlEvents:`をコントロール部品に送信することにより行います。ユーザが所定の方法でコントロール部品にタッチすると、コントロール部品はアクションメッセージを、`sendAction:to:from:forEvent:`メッセージの形でグローバルUIApplicationオブジェクトに転送します。AppKitと同様、グローバルアプリケーションオブジェクトは、アクションメッセージを一手に引き受け、ディスパッチする中枢として働きます。コントロール部品に、アクションメッセージのターゲットが`nil`と指定されていれば、アプリケーションはレスポンドチェーンをたどって、当該メッセージを処理できる（ターゲットセレクタに対応するメソッドを実装している）オブジェクトを検索します。

AppKitフレームワークではアクションメソッドのシグニチャが1種類、場合により2種類であるのに対し、UIKitフレームワークでは次の3通りのアクションセレクタが使えます。

```
- (void)action
```

- (void)action:(id)sender
- (void)action:(id)sender forEvent:(UIEvent \*)event

UIKitのターゲット-アクション機構について詳しくは、『*UIControl Class Reference*』を参照してください。

# フレームワークの相互乗り入れについて

Core FoundationフレームワークとFoundationフレームワークには、互いに入れ替えて使えるデータ型が多数あります。この性質をフレームワークの相互乗り入れ (*Toll-Free Bridging*) と言います。同じデータ型を、Core Foundationの関数呼び出しの引数としても、Objective-Cメッセージのレシーバとしても使えるのです。たとえばNSLocale (『*NSLocale Class Reference*』を参照) は、Core FoundationのCFLocale (『*CFLocale Reference*』を参照) と相互に入れ替えて使えます。したがって、あるメソッドの引数がNSLocale \*である場合、CFLocaleRef型の変数を渡してもよいことになります。同様に、引数型がCFLocaleRefであれば、NSLocaleのインスタンスを渡しても構いません。次の例のように型をキャストすれば、コンパイラの警告も出なくなります。

```
NSLocale *gbNSLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"en_GB"];
CFLocaleRef gbCFLocale = (CFLocaleRef) gbNSLocale;
CFStringRef cfIdentifier = CFLocaleGetIdentifier (gbCFLocale);
NSLog(@"cfIdentifier: %@", (NSString *)cfIdentifier);
// ログに「cfIdentifier: en_GB」と出力
CFRelease((CFLocaleRef) gbNSLocale);

CFLocaleRef myCFLocale = CFLocaleCopyCurrent();
NSLocale * myNSLocale = (NSLocale *) myCFLocale;
[myNSLocale autorelease];
NSString *nsIdentifier = [myNSLocale localeIdentifier];
CFShow((CFStringRef) [@"nsIdentifier: " stringByAppendingString:nsIdentifier]);
// 現ロケールの識別子をログ出力
```

この例から分かるように、メモリ管理関数/メソッドも互換性があります。CFReleaseにCocoaオブジェクトを渡し、あるいはCore Foundationオブジェクトをrelease、autoreleaseすることも可能です。

**注意** ガベージコレクションを使う場合、CocoaオブジェクトとCore Foundationオブジェクトでは、メモリ管理の方式に重要な差異があります。詳しくは“Using Core Foundation with Garbage Collection”を参照してください。

フレームワークの相互乗り入れが可能なのはMac OS X v10.0以降です。表 13-1に、Core FoundationとFoundationとで互換性のあるデータ型を示します。それぞれ、相互乗り入れ可能になったMac OS Xの版番号も載せてあります。

表 13-1 Core FoundationとFoundationとで互換性のあるデータ型

Core Foundationの型	Foundationのクラス	相互乗り入れ可能になった版
CFArrayRef	NSArray	Mac OS X v10.0
CFAttributedStringRef	NSAttributedString	Mac OS X v10.4
CFCalendarRef	NSCalendar	Mac OS X v10.4
CFCharacterSetRef	NSCharacterSet	Mac OS X v10.0
CFDataRef	NSData	Mac OS X v10.0
CFDateRef	NSDate	Mac OS X v10.0
CFDictionaryRef	NSDictionary	Mac OS X v10.0
CFErrorRef	NSError	Mac OS X v10.5
CFLocaleRef	NSLocale	Mac OS X v10.4
CFMutableArrayRef	NSMutableArray	Mac OS X v10.0
CFMutableAttributedStringRef	NSMutableAttributedString	Mac OS X v10.4
CFMutableCharacterSetRef	NSMutableCharacterSet	Mac OS X v10.0
CFMutableDataRef	NSMutableData	Mac OS X v10.0
CFMutableDictionaryRef	NSMutableDictionary	Mac OS X v10.0
CFMutableSetRef	NSMutableSet	Mac OS X v10.0
CFMutableStringRef	NSMutableString	Mac OS X v10.0
CFNumberRef	NSNumber	Mac OS X v10.0

Core Foundationの型	Foundationのクラス	相互乗り入れ可能になった版
CFReadStreamRef	NSInputStream	Mac OS X v10.0
CFRunLoopTimerRef	NSTimer	Mac OS X v10.0
CFSetRef	NSSet	Mac OS X v10.0
CFStringRef	NSString	Mac OS X v10.0
CFTimeZoneRef	NSTimeZone	Mac OS X v10.0
CFURLRef	NSURL	Mac OS X v10.0
CFWriteStreamRef	NSOutputStream	Mac OS X v10.0

**注意** 名前だけ見れば相互乗り入れ可能のようであっても、実際にはそうでないデータ型があります。たとえば、NSRunLoopとCFRunLoop、NSBundleとCFBundle、NSDateFormatterとCFDateFormatterは互換性がありません。

# 書類の改訂履歴

この表は「Objective-Cプログラミングの概念」の改訂履歴です。

日付	メモ
2012-01-09	Cocoa / Cocoa Touchに基づく開発において重要な、デザインパターン、アーキテクチャ、およびその他の考え方についての解説。

---





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

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

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

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

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

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

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

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

Apple, the Apple logo, Cocoa, Cocoa Touch, Mac, Mac OS, Objective-C, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

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

Apple Inc. は本書の内容を確認しておりますが、本書に関して、明示的であるか黙示的であるかを問わず、その品質、正確さ、市場性、または特定の目的に対する適合性に関して何らかの保証または表明を行うものではありません。その結果、本書は「現状有姿のまま」提供され、本書の品質または正確さに関連して発生するすべての損害は、購入者であるお客様が負うものとする。

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

上記の損害に対する保証および救済は、口頭や書面によるか、または明示的や黙示的であるかを問わず、唯一のものであり、その他一切の保証にかわる

ものです。Apple Inc. の販売店、代理店、または従業員には、この保証に関する規定に何らかの変更、拡張、または追加を加える権限は与えられていません。

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