
キー値コーディングプログラミングガイド

[Cocoa > Data Management](#)



2011-06-06



Apple Inc.
© 2003, 2011 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, AppleScript, Cocoa, Mac, Mac OS, and Objective-C are trademarks of Apple Inc., registered in the United States and other countries.

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

いかなる場合も、Apple Inc. は、本書の内容に含まれる瑕疵または不正確さによって生じる直接的、間接的、特殊的、偶発的、または結果的損害

に対する賠償請求には一切応じません。そのような損害の可能性があらかじめ指摘されている場合においても同様です。

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

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

目次

はじめに 7

この書類の構成 7

関連項目 8

キー値コーディングとは何か 9

キー値コーディングおよびスクリプティング機能 9

キー値コーディングによるコードの簡素化 10

用語の定義 13

キー値コーディングの基本 15

キーおよびキーパス 15

キー値コーディングによる値の取得 15

キー値コーディングによる値の設定 16

ドット構文とキー値コーディング 16

キー値コーディングのアクセサメソッド 19

アクセサに対してよく使用する命名パターン 19

対多プロパティに対するコレクション型アクセサの命名パターン 20

インデックス付きアクセサのパターン 21

順序なしアクセサのパターン 24

キー値の検証 27

検証メソッドの命名規則 27

検証メソッドの実装 27

検証メソッドの呼び出し 28

自動検証 28

スカラ値の検証 29

検証メソッドとメモリ管理 29

KVOの準拠 31

属性および対1リレーションシップのKVC互換性 31

インデックス付き対多リレーションシップのKVC互換性 31

順序なし対多リレーションシップのKVC互換性 32

スカラおよび構造体のサポート 33

非オブジェクト値の表現 33
nil値の処理 33
スカラ型のラップとその解除 34
構造体のラップとその解除 34

コレクション演算子 37

例に用いるデータ 37
単純型コレクション演算子 38
 @avg 38
 @count 39
 @max 39
 @min 39
 @sum 40
オブジェクト演算子 40
 @distinctUnionOfObjects 40
 @unionOfObjects 40
配列演算子、集合演算子 41
 @distinctUnionOfArrays 41
 @unionOfArrays 42
 @distinctUnionOfSets 42

アクセサ検索の実装の詳細 43

単純型属性に対するアクセサの検索パターン 43
 setValue:forKey:におけるデフォルトの検索パターン 43
 valueForKey:におけるデフォルトの検索パターン 43
順序付きコレクションに対するアクセサの検索パターン 44
一意順序付きコレクションに対するアクセサの検索パターン 45
順序なしコレクションに対するアクセサの検索パターン 46

プロパティのリレーションシップの記述 49

クラスの記述 49

パフォーマンスに関する考慮事項 51

キー値コーディングメソッドのオーバーライド 51
対多リレーションシップの最適化 51

書類の改訂履歴 53

図、表、リスト

キー値コーディングとは何か 9

- リスト 1 データソースメソッドの実装（キー値コーディングを使用していない場合） 10
- リスト 2 データソースメソッドの実装（キー値コーディングを使用した場合） 10

キー値コーディングのアクセサメソッド 19

- リスト 1 hiddenプロパティキーに対するアクセサの命名 19
- リスト 2 別の形式を使用した、hiddenプロパティキーに対するアクセサ 19
- リスト 3 hiddenプロパティキーをサポートするアクセサの命名規則 19
- リスト 4 -count<Key>の実装例 21
- リスト 5 -objectIn<Key>AtIndex:および-<key>AtIndexes:の実装例 21
- リスト 6 -get<Key>:range:の実装例 22
- リスト 7 アクセサ-insertObject:in<Key>AtIndex:および-insert<Key>:atIndexes:の実装例 23
- リスト 8 アクセサ-removeObjectFrom<Key>AtIndex:および-remove<Key>AtIndexes:の実装例 23
- リスト 9 アクセサ-replaceObjectIn<Key>AtIndex:withObject:および-replace<Key>AtIndexes:with<Key>:の実装例 24
- リスト 10 アクセサ-countOf<Key>、-enumeratorOf<Key>、-memberOf<Key>:の実装例 25
- リスト 11 アクセサ-add<Key>Object:および-add<Key>:の実装例 26
- リスト 12 アクセサ-remove<Key>Object:および-remove<Key>:の実装例 26
- リスト 13 -intersect<Key>:および-set<Key>:の実装例 26

キー値の検証 27

- リスト 1 nameプロパティに対する検証メソッドの宣言 27
- リスト 2 nameプロパティに対する検証メソッド 28
- リスト 3 スカラプロパティの検証メソッド 29
- リスト 4 validateName:error:の誤った呼び出し 30
- リスト 5 validateName:error:の正しい呼び出し 30

スカラおよび構造体のサポート 33

- 表 1 NSNumberオブジェクトにおいてラップされるスカラ型 34
- 表 2 NSValueを使用してラップされる一般的な構造体 34
- リスト 1 setNilValueForKey:の実装例 33

コレクション演算子 37

図 1	演算子とキーパスの書式 37
表 1	Transactionsオブジェクトのデータ例 38
表 2	配列moreTransactionsに含まれるTransactionのデータ 41

はじめに

この文書では、`NSKeyValueCoding`という簡易プロトコルについて解説します。アプリケーションがオブジェクトのプロパティに、名前（あるいはキー）を使って間接的にアクセスするための仕組みです。アクセサメソッドを起動する、あるいはインスタンス変数にアクセスする、という直接的な方法ではない点が重要です。

このキー値コーディングに従ってアプリケーションを記述する方法、そして、独自に開発するクラスをキー値コーディング互換にし、ほかのさまざまな技術と連携する方法について、この文書で理解してください。キー値コーディングは、キー値監視、Cocoaバインディング、Core Dataなどと連携し、AppleScriptに対応したアプリケーションを開発する上で、鍵となる技術です。この文書では、Cocoaを基盤とする開発の基本事項、特にObjective-C言語やメモリ管理については、よく知っているものと想定して解説します。

この書類の構成

『*Key-Value Coding*』は次の各章から成ります。

- 「[キー値コーディングとは何か](#)」（9 ページ）では、キー値コーディングの概要を紹介します。
- 「[用語の定義](#)」（13 ページ）では、オブジェクトのプロパティについて言及するときによく使用する用語の定義を取り上げます。
- 「[キー値コーディングの基本](#)」（15 ページ）では、キー値コーディングを使用するために必要となる、基本原則を説明します。
- 「[キー値コーディングのアクセサメソッド](#)」（19 ページ）では、クラスが実装する必要のあるアクセサメソッドについて説明します。
- 「[キー値の検証](#)」（27 ページ）では、プロパティの検証処理を実装する方法について説明します。
- 「[KVOの準拠](#)」（31 ページ）では、キー値コーディング互換にするためにクラスが実装しなければならない機能について説明します。
- 「[スカラおよび構造体のサポート](#)」（33 ページ）では、キー値コーディングがサポートするデータ型について説明します。
- 「[コレクション演算子](#)」（37 ページ）では、利用可能なコレクション演算子の一覧を示し、各演算子の使用方法について説明します。
- 「[アクセサ検索の実装の詳細](#)」（43 ページ）では、適切なアクセサメソッドまたはインスタンス変数を判断する方法について説明します。
- 「[プロパティのリレーションシップの記述](#)」（49 ページ）では、メタデータを使用して、オブジェクトとそのプロパティのリレーションシップを定義する方法について説明します。

- 「[パフォーマンスに関する考慮事項](#)」（51 ページ）では、キー値コーディングを使用するときにパフォーマンスに関して考慮すべき事柄について取り上げます。

関連項目

この文書では取り上げませんが、ほかにもキー値コーディングに関連する技術がいくつかあります。

- 『*Key-Value Observing Programming Guide*』では、オブジェクトがほかのオブジェクトの状態変化を監視するために用いる、キー値監視プロトコルについて解説しています。

キー値コーディングとは何か

キー値コーディングとは、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仕組みです。アクセサメソッドを呼び出してアクセスしたり、インスタンス変数としてアクセスするのではなく、プロパティの識別に文字列を使用してアクセスします。キー値コーディングでは、アクセサメソッドの実装パターンやシグニチャ（引数並び）の規約を定めています。

アクセサメソッドは、その名前が示すように、アプリケーションのデータモデルのプロパティ値にアクセスする手段を提供します。取得アクセサ、設定アクセサという、基本的な2種類のアクセサがあります。取得アクセサ（あるいはゲッター）は、プロパティ値を返します。設定アクセサ（あるいはセッター）は、プロパティ値を設定するために使います。ゲッターやセッターには、オブジェクトの属性や対多関係を扱う変種もあります。

キー値コーディング互換のアクセサを実装するか否かは、アプリケーション設計における重要な岐路になります。アクセサには、データの適切なカプセル化を促進し、メモリ管理をある1箇所に分離して集中的に行い、ほかのさまざまな技術（キー値監視、Core Data、Cocoaバインディング、スクリプトによる処理の記述など）とうまく統合できるようにする、という効用があります。キー値コーディングの手法は、多くの場合、アプリケーションのコードを簡素化するためにも使用できます。キー値コーディング互換のアクセサを実装し、これを使ってアプリケーションを構築する、という開発方針を推奨する理由については、『*Model Object Implementation Guide*』も参照してください。

キー値コーディングにおいて不可欠なメソッドは、`NSKeyValueCoding` Objective-C簡易プロトコルで宣言されており、デフォルトの実装は`NSObject`に用意されています。

キー値コーディングは、値がオブジェクトであるプロパティ、スカラ型、構造体もサポートします。オブジェクトでないパラメータと戻り型が検出されると自動的にラップされ、必要に応じてラップが解除されます（「[スカラおよび構造体のサポート](#)」（33 ページ）を参照）。

キー値コーディングおよびスクリプティング機能

Cocoa におけるスクリプティング機能は、アプリケーションにおいてモデルオブジェクトを通じてスクリプティングを簡単に実装できるように設計されています。モデルオブジェクトとは、アプリケーションのデータをカプセル化するオブジェクトのことです（『*Model Object Implementation Guide*』を参照）。ユーザがアプリケーションを対象に `AppleScript` コマンドを実行するとき、その目標は、コマンドを通してアプリケーションのモデルオブジェクトに直接アクセスし、処理を実行することです。

Mac OS X におけるスクリプティング機能は、`AppleScript` コマンドの実行を自動的にサポートするため、キー値コーディングに大きく依存しています。スクリプト対応のアプリケーションでは、モデルオブジェクトが、サポートするキーのセットを定義します。個々のキーは、モデルオブジェクトのプロパティを表します。スクリプティング関連のキーの具体例としては、`words`、`font`、`documents`、`color`などが挙げられます。キー値コーディングのAPIは、オブジェクトのキーの値をオブジェクトに対して問い合わせたり、オブジェクトのキーに新しい値を設定したりする、自動化された汎用の方法を提供します。

アプリケーションのオブジェクトを設計する際には、モデルオブジェクトのためのキーのセットを定義し、キーに対応するアクセサメソッドを実装する必要があります。そうしておけば、アプリケーションのスクリプトスイートを定義するときには、スクリプト対応の各クラスがサポートしているキーを指定できます。キー値コーディングをサポートすれば、スクリプティング機能に対するサポートも「何もしなくても」豊富に手に入ります。

AppleScriptでは、オブジェクト階層が、アプリケーションで使用されるモデルオブジェクトの構造を定義します。AppleScriptコマンドの大半は、親コンテナから子要素に向かってオブジェクト階層をたどることによってアプリケーション内の1つ以上のオブジェクトを指定します。クラスの記述において、利用可能なプロパティ間のリレーションシップをキー値コーディングによって定義できます。詳細については、「[プロパティのリレーションシップの記述](#)」（49 ページ）を参照してください。

Cocoaのスクリプティング機能は、キー値コーディングを利用して、スクリプト対応オブジェクトの情報の取得や設定を行います。Objective-C簡易プロトコルであるNSScriptKeyValueCodingのメソッドは、その他にも、キー値コーディングで活用できる機能を提供します。たとえば、複数の値を持つキーでのインデックスを使用したキー値の取得や設定、適切なデータ型へのキー値の変換などの機能を利用できます。

キー値コーディングによるコードの簡素化

作成するコード内でキー値コーディングを使用することで、実装を一般化できます。たとえば、NSTableViewオブジェクトおよびNSOutlineViewオブジェクトはいずれも、それぞれのカラム1つ1つに識別子文字列を関連付けます。この識別子を、表示したいプロパティのキーと同じにすれば、コードを大幅に簡素化できます。

リスト1は、キー値コーディングを使用しない場合のNSTableView デリゲートメソッドの実装です。リスト2は、キー値コーディングを使用した実装で、キーの代わりにカラム識別子を使用して適切な値を返します。

リスト1 データソースメソッドの実装（キー値コーディングを使用していない場合）

```
- (id)tableView:(NSTableView *)tableView
    objectValueForTableColumn:(id)column
    row:(int)row
{
    ChildObject *child = [childrenArray objectAtIndex:row];
    if ( [[column identifier] isEqualToString:@"name"] ) {
        return [child name];
    }
    if ( [[column identifier] isEqualToString:@"age"] ) {
        return [child age];
    }
    if ( [[column identifier] isEqualToString:@"favoriteColor"] ) {
        // etc...
    }
    // etc...
}
```

リスト2 データソースメソッドの実装（キー値コーディングを使用した場合）

```
- (id)tableView:(NSTableView *)tableView
    objectValueForTableColumn:(id)column
```

```
        row:(int)row  
{  
    ChildObject *child = [childrenArray objectAtIndex:row];  
    return [child valueForKey:[column identifier]];  
}
```


用語の定義

キー値コーディングでは、通常用語に別の意味を持たせたり、ほかにはない独特の用語を導入したりしています。

キー値コーディングは、属性、対1リレーションシップ、そして対多リレーションシップという3種類のオブジェクト値にアクセスするために使用できます。**プロパティ**という用語は、これらのオブジェクト値のことを指しています。

属性は、スカラや文字列、ブール値など、単純な値であるプロパティです。NSColorやNSNumberといった不変オブジェクトも属性とみなされます。

対1リレーションシップを指定するプロパティとは、固有のプロパティを持つオブジェクトのことです。これらの基本的なプロパティは、オブジェクト自体が変化しなくても、変化することが可能です。たとえば、NSViewインスタンスのスーパービューは、対1リレーションシップです。

最後に、**対多リレーションシップ**を指定するプロパティは、相互に関連するオブジェクトのコレクションで構成されます。このようなコレクションの保持には、NSArrayやNSSetのインスタンスがよく使用されます。ただし、キー値コーディングでは、カスタムのクラスもコレクションとして使用することができます。その場合、[「対多プロパティに対するコレクション型アクセサの命名パターン」](#)（20 ページ）の説明のようにキー値コーディングアクセサを実装すれば、コレクションがNSArrayやNSSetであるかのようにアクセスできます。

キー値コーディングの基本

この記事では、キー値コーディングの基本原則について説明します。

キーおよびキーパス

キーとは、オブジェクトの特定のプロパティを識別する文字列です。通常、キーは、受け取り側のオブジェクトのアクセサメソッドかインスタンス変数の名前と同じです。キーは、ASCIIエンコードを使用し、最初の文字は小文字でなければならず、空白を入れることはできません。

キーの具体例としては、`payee`、`openingBalance`、`transactions`、`amount`などが挙げられます。

キーパスとは、ドットで区切られた複数のキーからなる文字列のことで、オブジェクトプロパティを渡り歩くためのシーケンスの指定に使用します。シーケンス内の最初のキーのプロパティはレシーバを起点として相対的に決まり、その後続くキーは、1つ前のプロパティの値に対する相対的な値として評価されます。

たとえば、`address.street`というキーパスでは、`address`プロパティの値はレシーバオブジェクトから取得され、`street`プロパティは`address`オブジェクトに基づいて相対的に決まります。

キー値コーディングによる値の取得

`valueForKey:`メソッドは、レシーバを起点として相対指定したキーに対応する値を返します。指定したキーについてアクセサまたはインスタンス変数が存在しない場合、レシーバは自身に対して`valueForKeyUndefinedKey:`メッセージを送信します。`valueForKeyUndefinedKey:`のデフォルトの実装では、`NSUndefinedKeyException`が発生しますが、サブクラスはこの動作をオーバーライドできます。

同様に、`valueForKeyPath:`メソッドは、レシーバを起点として相対指定したキーパスに対応する値を返します。キーパスシーケンス内のオブジェクトで、該当するキーについてキー値コーディング互換でないオブジェクトは、`valueForKeyUndefinedKey:`メッセージを受け取ります。

`dictionaryWithValuesForKeys:`メソッドは、レシーバを起点として相対指定したキーの配列に対応する、それぞれの値を取得します。返された`NSDictionary`には、配列内のすべてのキーの値が格納されています。

注意： コレクションオブジェクト（NSArrayやNSSet、NSDictionaryなど）は、値としてnilを格納することができません。その代わり、nilの値は、NSNullという特別なオブジェクトによって表現されます。NSNullは、オブジェクトのプロパティのnil値を表す、単一のインスタンスを提供します。dictionaryWithValuesForKeys:およびsetValuesForKeysWithDictionary:のデフォルトの実装では、NSNullとnilの間の変換が自動的に行われます。したがって、オブジェクトが値がNSNullかどうかを明示的にテストする必要はありません。

対多プロパティのキーを含んだキーパスに対する値が返されたとき、そのキーがパスの中の最後のキーではない場合、返された値は、対多キーの右側のキーに対応するすべての値が格納されているコレクションです。たとえば、transactions.payeeというキーパスの値を要求すると、すべてのトランザクションについて、すべてのpayeeオブジェクトが格納されている配列が返されます。これは、キーパスに複数の配列が含まれる場合も同様です。accounts.transactions.payeeというキーパスでは、すべてのアカウントのすべてのトランザクションについて、すべての受取人オブジェクトが格納されている配列が返されます。

キー値コーディングによる値の設定

setValue:forKey:メソッドは、レシーバを起点として相対指定したキーに対応する値を、指定した値に設定します。setValue:forKey:のデフォルトの実装では、スカラや構造体を表すNSValueオブジェクトのラップを自動的に解除し、プロパティに代入するようになっています。ラップおよびその解除については、「スカラおよび構造体のサポート」（33ページ）で詳しく説明しています。

指定したキーが存在しない場合、レシーバにはsetValue:forUndefinedKey:メッセージが送信されます。setValue:forUndefinedKey:のデフォルトの実装では、NSUndefinedKeyExceptionが発生しますが、サブクラスはこのメソッドをオーバーライドして、独自の方法で要求を処理できます。

setValue:forKeyPath:メソッドも同じように動作しますが、単一のキーのほか、キーパスを扱うことができます。

最後に、setValuesForKeysWithDictionary:は、プロパティの識別に辞書キーを使用して、指定した辞書内の値でレシーバのプロパティを設定します。デフォルトの実装では、キーと値のペアごとにsetValue:forKey:を呼び出し、必要に応じてNSNullオブジェクトをnilで置き換えます。

もう1つ考慮すべきことは、オブジェクトでないプロパティをnil値に設定しようとした場合に何が起きるかということです。この場合、レシーバは自身に対してsetNilValueForKey:メッセージを送信します。setNilValueForKey:のデフォルトの実装では、NSInvalidArgumentExceptionが発生します。アプリケーションでは、このメソッドをオーバーライドしてデフォルト値かマーカ値に置き換え、新しい値を指定してsetValue:forKey:を呼び出すことができます。

ドット構文とキー値コーディング

Objective-C2.0のドット構文とキー値コーディングは、相互に独立した技術です。ドット構文を使用するかどうかに関係なくキー値コーディング（KVC）を使用でき、KVCを使用するかどうかに関係なくドット構文を使用できます。どちらも「ドット構文」を使用できます。キー値コーディングの場合、構文を使ってキーパス内の要素を区切ります。重要なのは、ドット構文でプロパティにアクセスする場合でも、実際にはレシーバの標準アクセサメソッドを起動している、ということです。

たとえば、次のように定義されたクラスの場合、キー値コーディングメソッドを使ってプロパティにアクセスできます。

```
@interface MyClass
@property NSString *stringProperty;
@property NSInteger integerProperty;
@property MyClass *linkedInstance;
@end
```

KVCを使ってインスタンス内のプロパティにアクセスできます。

```
MyClass *myInstance = [[MyClass alloc] init];
NSString *string = [myInstance valueForKey:@"stringProperty"];
[myInstance setValue:[NSNumber numberWithInt:2] forKey:@"integerProperty"];
```

プロパティのドット構文とKVCキープスの違いを示すために、次の例を考えてみましょう。

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
myInstance.linkedInstance.integerProperty = 2;
```

これは次のコードと同じ結果になります。

```
MyClass *anotherInstance = [[MyClass alloc] init];
myInstance.linkedInstance = anotherInstance;
[myInstance setValue:[NSNumber numberWithInt:2]
                 forKeyPath:@"linkedInstance.integerProperty"];
```


キー値コーディングのアクセサメソッド

`valueForKey:`、`setValueForKey:`、`mutableArrayValueForKey:`、`mutableSetValueForKey:`の呼び出しに使用するアクセサメソッドをキー値コーディングで指定するには、キー値コーディングの規約に則ったアクセサメソッドを実装する必要があります。

注意： この節ではアクセサのパターンを、「`-set<Key>:`」、あるいは「`-<key>`」というような形で表記します。ここで「`<key>`」の部分は、実際のプロパティ名に置き換えます。対応するメソッドは、「`<Key>`」または「`<key>`」の部分をプロパティ名で置き換えた名前を実装します（プロパティ名の先頭を、前者ならば大文字、後者ならば小文字に変更）。たとえば「`name`」というプロパティの場合、「`-set<Key>:`」というパターンを置き換えると「`-setName:`」、同様に「`-<key>`」は「`-name`」となります。

アクセサに対してよく使用する命名パターン

プロパティを返すアクセサメソッドの名前は`-<key>`という形式です。`-<key>`メソッドは、オブジェクト、スカラ、またはデータ構造体を返します。ブール型のプロパティについては、`-is<Key>`という別の形式の名前もサポートされています。

リスト 1 の例は、通常の命名規則を使用した、`hidden`プロパティに対するメソッドの宣言です。リスト 2 は、別の形式を使用した例です。

リスト 1 hiddenプロパティキーに対するアクセサの命名

```
- (BOOL)hidden
{
    // 実装固有のコード
    return ...;
}
```

リスト 2 別の形式を使用した、hiddenプロパティキーに対するアクセサ

```
- (BOOL)isHidden
{
    // 実装固有のコード
    return ...;
}
```

属性または対1リレーションシップのプロパティで`setValueForKey:`をサポートするには、`set<Key>:`という形式の名前のアクセサを実装する必要があります。リスト 3 は、`hidden`プロパティキーに対するアクセサメソッドです。

リスト 3 hiddenプロパティキーをサポートするアクセサの命名規則

```
- (void)setHidden:(BOOL)flag
```

```
{
    // 実装固有のコード
    return;
}
```

属性が非オブジェクト型であれば、nil値を表す適切な手段も実装しなければなりません。属性にnilを設定しようとする、キー値コーディングメソッドsetNilValueForKey:が呼び出されます。これをフックとして用い、適切なデフォルト値を設定したり、対応するアクセサが当該クラスに実装されていないキーを処理したりすることができます。

hidden属性にnilを設定しようとしたとき、代わりにYESが設定されるようにした例を示します。ブール値を収容するNSNumberのインスタンスを生成し、setValue:forKey:を使って新しい値を設定しています。このようにモデルをカプセル化すると、ある値を設定する際に必要な処理（アクション）がある場合に、その記述を書き漏らす危険がなくなります。したがって、アクセサメソッドを呼び出す、あるいはインスタンス変数に値を直接代入するよりも、優れた方法と言えるでしょう。

```
- (void)setNilValueForKey:(NSString *)theKey
{
    if ([theKey isEqualToString:@"hidden"]) {
        [self setValue:[NSNumber numberWithInt:YES] forKey:@"hidden"];
    } else
        [super setNilValueForKey:theKey];
}
```

対多プロパティに対するコレクション型アクセサの命名パターン

対多リレーションシップのプロパティに対するアクセサメソッドは、-<key>および-set<Key>:という形式のアクセサ名で実装することもできますが、これは一般に、コレクションオブジェクトを作成するためだけに使ってください。コレクションの要素を操作するためには、別途「コレクション型アクセサ」というメソッドを実装するよう推奨します。実装したこのメソッドを直接使うか、あるいは、mutableArrayValueForKey:またはmutableSetValueForKey:から返される可変コレクションプロキシを使います。

リレーションシップを取得する基本的なゲッタの代わりに、あるいはこれに追加して、コレクション型アクセサメソッドを実装すると、次のような利点があります。

- 可変対多リレーションシップに関しては、多くの場合、性能を大幅に改善できます。
- 適切なコレクション型アクセサを実装することにより、対多リレーションシップをNSArrayやNSSet以外のクラスでモデル化できます。プロパティのコレクション型アクセサメソッドを実装すれば、キー値コーディングメソッドを使う際、配列や集合と同じようにプロパティを扱うようになります。
- コレクション型アクセサメソッドを直接使って、キー値監視に適合した方法で、コレクションに修正を施すことができます。キー値監視については『*Key-Value Observing Programming Guide*』を参照してください。

コレクション型アクセサは2種類あります。順序付き対多リレーションシップ（一般にNSArrayで表す）に用いるインデックス付きアクセサと、要素間の順序が定義されないリレーションシップ（一般にNSSetで表す）に用いる順序なしアクセサです。

インデックス付きアクセサのパターン

インデックス付きアクセサメソッドは、順序付きリレーションシップにおけるオブジェクトを数え、検索し、追加し、置換するための仕組みを定義します。一般にこのリレーションシップは、NSArrayやNSMutableArrayのインスタンスです。しかし以下のメソッドを定義すれば、どのようなオブジェクトでも配列と同じように操作できます。実装したメソッドを実際を使って、リレーションシップで結ばれたオブジェクトを直接操作することも可能です。

コレクションからデータを取得するインデックス付きアクセサ（ゲッタの異型）と、mutableArrayValueForKey:のインターフェイスとして動作し、コレクションに変更を施す可変アクセサがあります。

インデックス付きゲッタ

順序付き対多リレーションシップを対象とする読み込み専用のアクセス手段として、次のメソッドを実装してください。

- -countOf<Key>。必須。NSArrayに組み込みのメソッドcountと同様の機能です。
- -objectIn<Key>AtIndex:または-<key>AtIndexes:。いずれか一方のみを実装してください。NSArrayのobjectAtIndex:メソッド、objectsAtIndexes:メソッドにそれぞれ相当します。
- -get<Key>:range:。必須ではありませんが、実装すれば性能を改善できます。NSArrayのgetObjects:range:メソッドに相当します。

-countOf<Key>メソッドは、対多リレーションシップに属するオブジェクトの個数を、NSUInteger型の値として返すように実装します。リスト 4に、対多リレーションシップemployeesの-countOf<Key>メソッドの実装例を示します。

リスト 4 -count<Key>の実装例

```
- (NSUInteger)countOfEmployees
{
    return [employees count];
}
```

-objectIn<Key>AtIndex:メソッドは、対多リレーションシップに属する、指定されたインデックスのオブジェクトを返すように実装します。-<key>AtIndexes:メソッドは、NSIndexSetパラメータで指定されたインデックス（複数）のオブジェクトを、配列として返すように実装します。いずれか一方のみを実装してください。

リスト 5に、対多リレーションシップemployeesの-objectIn<Key>AtIndex:メソッド、-<key>AtIndexes:メソッドの実装例を示します。

リスト 5 -objectIn<Key>AtIndex:および-<key>AtIndexes:の実装例

```
- (id)objectInEmployeesAtIndex:(NSUInteger)index
{
    return [employees objectAtIndex:index];
}
```

```

        return [employees objectAtIndex:index];
    }

    - (NSArray *)employeesAtIndexes:(NSIndexSet *)indexes
    {
        return [employees objectAtIndexes:indexes];
    }

```

ベンチマークの結果、処理性能を改善する必要があると判断した場合は、`-get<Key>:range:`を実装するとよいでしょう。第2引数で指定された範囲のオブジェクトを、第1引数のバッファに収容して返すように実装します。

リスト 6に、対多リレーションシップemployeesの`-get<Key>:range:`メソッドの実装例を示します。

リスト 6 `-get<Key>:range:`の実装例

```

- (void)getEmployees:(Employee **)buffer range:(NSRange)inRange
{
    // 指定された範囲のオブジェクトを、与えられたバッファに
    // 収容して返す。次の例は、employeesがNSArrayに格納されていると
    // 想定した実装
    [employees getObjects:buffer range:inRange];
}

```

可変インデックス付きアクセサ

可変対多リレーションシップをインデックス付きアクセサで操作できるようにするためには、ほかにいくつかメソッドを実装する必要があります。可変インデックス付きアクセサを実装すると、`mutableArrayValueForKey:`から返される配列プロキシを使って、簡便かつ効率的に、インデックス付きコレクションを操作できるようになります。さらに、対多リレーションシップにこのメソッドを実装すれば、クラスは当該プロパティに関してキー値監視互換になります（『*Key-Value Observing Programming Guide*』を参照）。

注意： ここで説明する可変アクセサをぜひ実装して使うよう推奨します。可変集合を直接返すアクセサは、できるだけ使わないようにしてください。リレーションシップのデータに変更を施す場合、可変アクセサは非常に効率的な手段となります。

可変順序付き対多リレーションシップをキー値コーディング互換にするためには、次のメソッドを実装する必要があります。

- `-insertObject:in<Key>AtIndex:`または`-insert<Key>:atIndexes:`。いずれか一方のみを実装してください。NSMutableArrayの`insertObjectAtIndex:`メソッド、`insertObjects:atIndexes:`メソッドにそれぞれ相当します。
- `-removeObjectFrom<Key>AtIndex:`または`-remove<Key>AtIndexes:`。いずれか一方のみを実装してください。NSMutableArrayの`removeObjectAtIndex:`メソッド、`removeObjectsAtIndexes:`メソッドにそれぞれ対応します。
- `-replaceObjectIn<Key>AtIndex:withObject:`または`-replace<Key>AtIndexes:with<Key>:`。必須ではありません。ベンチマークの結果、処理性能に問題がある場合のみ実装してください。

-insertObject:in<Key>AtIndex:メソッドには、挿入するオブジェクトと、挿入位置を表すNSUInteger型のインデックスを指定します。-insert<Key>:atIndexes:メソッドは、配列で指定された各オブジェクトを、コレクションの、NSIndexSetで指定されたインデックス位置にそれぞれ挿入します。いずれか一方のみを実装してください。

リスト7に、対多プロパティemployeeの挿入アクセサの実装例（2通り）を示します。

リスト7 アクセサ-insertObject:in<Key>AtIndex:および-insert<Key>:atIndexes:の実装例

```
- (void)insertObject:(Employee *)employee
    inEmployeesAtIndex:(NSUInteger)index
{
    [employees insertObject:employee atIndex:index];
    return;
}

- (void)insertEmployees:(NSArray *)employeeArray
    atIndexes:(NSIndexSet *)indexes
{
    [employees insertObjects:employeeArray atIndexes:indexes];
    return;
}
```

-removeObjectFrom<Key>AtIndex:メソッドには、リレーションシップから削除するオブジェクトの位置を表す、NSUInteger型のインデックスを指定します。-remove<Key>AtIndexes:には、リレーションシップから削除するオブジェクト（複数）のインデックスを、NSIndexSet型の値で指定します。これも同様に、いずれか一方のみを実装してください。

リスト8に、対多プロパティemployeeを対象とする-removeObjectFrom<Key>AtIndex:および-remove<Key>AtIndexes:の実装例を示します。

リスト8 アクセサ-removeObjectFrom<Key>AtIndex:および-remove<Key>AtIndexes:の実装例

```
- (void)removeObjectFromEmployeesAtIndex:(NSUInteger)index
{
    [employees removeObjectAtIndex:index];
}

- (void)removeEmployeesAtIndexes:(NSIndexSet *)indexes
{
    [employees removeObjectsAtIndexes:indexes];
}
```

ベンチマークの結果、処理性能を改善する必要があると判断した場合は、置換アクセサ（一方または両方）を実装するとよいでしょう。-replaceObjectIn<Key>AtIndex:withObject:や-replace<Key>AtIndexes:with<Key>:は、コレクション中のオブジェクトを置換するメソッドです。これが実装されていれば、削除メソッドと挿入メソッドを組み合わせる呼び出す必要がありません。

リスト9に、対多プロパティemployeeを対象とする、-replaceObjectIn<Key>AtIndex:withObject:および-replace<Key>AtIndexes:with<Key>:の実装例を示します。

リスト 9 アクセサ-replaceObjectIn<Key>AtIndex:withObject:および-replace<Key>AtIndexes:with<Key>:の実装例

```
- (void)replaceObjectInEmployeesAtIndex:(NSUInteger)index
    withObject:(id)anObject
{
    [employees replaceObjectAtIndex:index withObject:anObject];
}

- (void)replaceEmployeesAtIndexes:(NSIndexSet *)indexes
    withEmployees:(NSArray *)employeeArray
{
    [employees replaceObjectsAtIndexes:indexes
    withObject:employeeArray];
}
```

順序なしアクセサのパターン

順序なしアクセサメソッドは、順序なしリレーションシップに属するオブジェクトにアクセスし、変更を施すための仕組みを与えます。一般にこのリレーションシップは、NSSetやNSMutableSetのインスタンスです。しかし以下のアクセサを実装すれば、どのようなクラスでも、NSSetのインスタンスと同じようにリレーションシップをモデル化し、キー値コーディングで操作できるようになります。

順序なしリレーションシップの取得アクセサ

順序なしリレーションシップを対象とする取得アクセサも、リレーションシップのデータにアクセスするための仕組みを定義します。コレクション中のオブジェクトの個数を返す、各オブジェクトについて所定の処理を繰り返す、コレクションに属するオブジェクトを比較して、あるオブジェクトが要素として含まれているか否かを判定する、というメソッドがあります。

注意： 順序なしの取得アクセサを、実際に実装する機会はほとんどないでしょう。順序なしの対多リレーションシップは、多くの場合、NSSetまたはそのサブクラスのインスタンスとしてモデル化します。この場合、プロパティのアクセサパターンに当てはまるメソッドが実装されていなくても、キー値コーディングにより直接集合にアクセスできます。一般に、実装が必要なのは、独自のコレクションクラスに対して集合と同じようにアクセスしたい場合だけです。

順序なしの対多リレーションシップを対象とする読み込み専用のアクセス手段として、次のメソッドを実装してください。

- -countOf<Key>。必須。NSSetのcountメソッドに相当します。
- -enumeratorOf<Key>。必須。NSSetのobjectEnumeratorメソッドに相当します。
- -memberOf<Key>:。必須。NSSetのmember:メソッドに相当します。

リスト 10に、上記の取得アクセサの実装例を示します。単にtransactionsプロパティに委譲しているだけの実装です。

リスト 10 アクセサ-countOf<Key>、-enumeratorOf<Key>、-memberOf<Key>:の実装例

```
- (NSUInteger)countOfTransactions
{
    return [transactions count];
}

- (NSEnumerator *)enumeratorOfTransactions
{
    return [transactions objectEnumerator];
}

- (Transaction *)memberOfTransactions:(Transaction *)anObject
{
    return [transactions member:anObject];
}
```

アクセサ-countOf<Key>は、リレーションシップの項目数を返すように実装します。
-enumeratorOf<Key>メソッドは、リレーションシップの各項目について処理を繰り返すために用いる、NSEnumeratorのインスタンスを返します。列挙子について詳しくは、『*Collections Programming Topics*』の「Enumeration: Traversing a Collection's Elements」を参照してください。

アクセサ-memberOf<Key>:は、引数として渡されたオブジェクトをコレクションの各要素と比較し、一致したオブジェクト（ない場合はnil）を返します。オブジェクトの比較にはisEqual:を使うのが普通ですが、ほかの方法で比較することもできます。返すオブジェクトは、比較に用いたものでなくても、内容が同等であれば構いません。

順序なしリレーションシップの可変アクセサ

順序なしの対多リレーションシップに対して修正を施すためには、さらにいくつかメソッドを実装しなければなりません。順序なしリレーションシップの可変アクセサを実装すると、mutableSetValueForKey:から返される配列プロキシを使って、簡便かつ効率的に、コレクションを操作できるようになります。さらに、対多リレーションシップにこのメソッドを実装すれば、クラスは当該プロパティに関してキー値監視互換になります（『*Key-Value Observing Programming Guide*』を参照）。

注意： ここで説明する可変アクセサをぜひ実装して使うよう推奨します。可変配列を直接返すアクセサは、できるだけ使わないようにしてください。リレーションシップのデータに変更を施す場合、可変アクセサは非常に効率的な手段となります。

可変順序なし対多リレーションシップをキー値コーディング互換にするためには、次のメソッドを実装する必要があります。

- -add<Key>Object:または-add<Key>:. いずれか一方のみを実装してください。NSMutableSetのaddObject:メソッドに相当します。
- -remove<Key>Object:または-remove<Key>:. いずれか一方のみを実装してください。NSMutableSetのremoveObject:メソッドに相当します。
- -intersect<Key>:. 必須ではありません。ベンチマークの結果、処理性能に問題がある場合のみ実装してください。NSSetのintersectSet:メソッドに相当します。

`-add<Key>Object:`と`-add<Key>:`は、単一の項目または集合をリレーションシップに追加します。いずれか一方のみを実装してください。複数の項目（集合）を追加する場合、同等のオブジェクトが重複しないようにしなければなりません。リスト 11に実装例を示します。単に`transactions`プロパティに委譲しているだけの実装です。

リスト 11 アクセサ`-add<Key>Object:`および`-add<Key>:`の実装例

```
- (void)addTransactionsObject:(Transaction *)anObject
{
    [transactions addObject:anObject];
}

- (void)addTransactions:(NSSet *)manyObjects
{
    [transactions unionSet:manyObjects];
}
```

同様に`-remove<Key>Object:`と`-remove<Key>:`は、単一の項目または集合をリレーションシップから削除します。これも同様に、いずれか一方のみを実装してください。リスト 12に実装例を示します。単に`transactions`プロパティに委譲しているだけの実装です。

リスト 12 アクセサ`-remove<Key>Object:`および`-remove<Key>:`の実装例

```
- (void)removeTransactionsObject:(Transaction *)anObject
{
    [transactions removeObject:anObject];
}

- (void)removeTransactions:(NSSet *)manyObjects
{
    [transactions minusSet:manyObjects];
}
```

ベンチマークの結果、処理性能を改善する必要があると判断した場合は、`-intersect<Key>:`または`-set<Key>:`も実装するとよいでしょう（リスト 13を参照）。

`-intersect<Key>:`は、一方の集合にしかないオブジェクトを、リレーションシップからすべて削除します。`NSMutableSet`の`intersectSet:`メソッドに相当します。

リスト 13 `-intersect<Key>:`および`-set<Key>:`の実装例

```
- (void)intersectTransactions:(NSSet *)otherObjects
{
    return [transactions intersectSet:otherObjects];
}
```

キー値の検証

キー値コーディングには、プロパティ値を検証するための一貫性のあるAPIが備わっています。検証のインフラストラクチャは、クラスが値を受け付け、代替値を提供したり、プロパティの新しい値を拒否してエラーの理由を示したりする機会を提供します。

検証メソッドの命名規則

アクセサメソッドに命名規則があるのと同様、プロパティの検証メソッドにも命名規則があります。検証メソッドでは、`validate<Key>:error:`という形式を使用します。リスト1の例は、`name`というプロパティに対する検証メソッドの宣言です。

リスト1 nameプロパティに対する検証メソッドの宣言

```
-(BOOL)validateName:(id *)ioValue error:(NSError **)outError {  
    // 実装固有のコード  
    return ...;  
}
```

検証メソッドの実装

検証メソッドには、2つのパラメータが参照渡しされます。1つは検証対象の値オブジェクト、もう1つはエラー情報を返すために使用するNSErrorです。

検証メソッドから得られる結果として、次の3つが考えられます。

1. オブジェクト値は有効であり、YESが返されます。値オブジェクトおよびエラーは変更されません。
2. オブジェクト値が無効で、有効な値を作成して返すことができません。この場合は、エラーパラメータが、検証に失敗した理由を示すNSErrorオブジェクトに設定された後に、NOが返されます。
3. 有効なオブジェクト値を新たに作成して返します。この場合、値パラメータが新たに作成された有効な値に設定された後にYESが返されます。エラーは、変更されずに返されます。渡された`ioValue`が可変であっても、それを修正するのではなく、新しいオブジェクトを返さなければなりません。

一般に、「検証」メソッドでこのように新たに値を設定することは推奨できません。値が整合性を保って伝搬することの保証が難しく、メモリ管理にも問題が生じる可能性があるからです（「[検証メソッドとメモリ管理](#)」（29 ページ）を参照）。

リスト 2 の例は、name プロパティについて、値オブジェクトが文字列であり、名前の大文字小文字が正しく表記されていることを確認する検証メソッドを実装します。


リスト 2 name プロパティに対する検証メソッド

```
-(BOOL)validateName:(id *)ioValue error:(NSError **)outError
{
    // nameがnilであってはならず、また、2文字以上でなければならない。
    if ((*ioValue == nil) || ([NSString *)ioValue length] < 2)) {
        if (outError != NULL) {
            NSString *errorString = NSLocalizedStringFromTable(
                @"A Person's name must be at least two characters long",
                @"Person",
                @"validation: too short name error");
            NSDictionary *userInfoDict =
                [NSDictionary dictionaryWithObject:errorString
                                     forKey:NSLocalizedStringDescriptionKey];
            *outError = [[[NSError alloc] initWithDomain:PERSON_ERROR_DOMAIN
                                     code:PERSON_INVALID_NAME_CODE
                                     userInfo:userInfoDict]
                autorelease];
        }
        return NO;
    }
    return YES;
}
```

重要： 検証の結果NOを返すと判断した場合、*outError*パラメータがNULLでないことをまず確認しなければなりません。次いで、*outError*パラメータに適切なNSErrorオブジェクトを設定し、NOを返します。

検証メソッドの呼び出し

検証メソッドは、直接呼び出すことも、*validateValue:forKey:error:*を呼び出してキーを指定することもできます。*validateValue:forKey:error:*のデフォルトの実装は、*validate<Key>:error:*というパターンに名前が一致する検証メソッドをレシーバのクラスの中で検索します。該当するメソッドが見つかり、そのメソッドが呼び出され、結果が返されます。該当するメソッドが存在しなければ、*validateValue:forKey:error:*はYESを返し、値を有効にします。

 **警告：** プロパティに対する*-set<Key>:*の実装から決して検証メソッドを呼び出してはなりません。

自動検証

キー値コーディングでは、検証は自動的に実行されません。アプリケーション内で検証メソッドを呼び出すようにします。

いくつかのCocoaバインディングでは、自動的に検証するよう指定できます。詳しくは『*CocoaBindings Programming Topics*』を参照してください。Core Dataは、管理オブジェクトコンテキストを保存する際、自動的に検証を行います。

スカラ値の検証

検証メソッドは、値パラメータがオブジェクトであると想定します。したがって、オブジェクトでないプロパティの値は「[スカラおよび構造体のサポート](#)」（33ページ）で説明するようにNSValueオブジェクトまたはNSNumberオブジェクトにラップされます。リスト 3の例は、ageというスカラプロパティを検証するメソッドです。

リスト 3 スカラプロパティの検証メソッド

```
-(BOOL)validateAge:(id *)ioValue error:(NSError **)outError
{
    if (*ioValue == nil) {
        // これをsetNilValueForKeyでトラップ
        // ほかに、ここで新規に値が0のNSNumberを作成するという方法もある
        return YES;
    }
    if ([*ioValue floatValue] <= 0.0) {
        NSString *errorString = NSLocalizedStringFromTable(
            @"Age must be greater than zero", @"Person",
            @"validation: zero age error");
        NSDictionary *userInfoDict =
            [NSDictionary dictionaryWithObject:errorString
            forKey:NSLocalizedStringKey];
        NSError *error = [[NSError alloc] initWithDomain:PERSON_ERROR_DOMAIN
            code:PERSON_INVALID_AGE_CODE
            userInfo:userInfoDict] autorelease];
        *outError = error;
        return NO;
    }
    else {
        return YES;
    }
    // ...
}
```

検証メソッドとメモリ管理

検証メソッドは、参照渡しされる元の値オブジェクトとエラーオブジェクトの両方を置き換えることができるため、メモリ管理を正しく行うために十分注意しなければなりません。アプリケーションが作成して検証メソッドにパラメータとして渡すオブジェクトは、検証メソッドを呼び出す前に自動解放されるよう、アプリケーションを設計する必要があります。

たとえば、リスト 4のコードはリスト 2に示したvalidateName:error:メソッドを呼び出します。このコードは、newNameオブジェクトを作成し、自動解放しないままvalidateName:error:メソッドに、作成したオブジェクトを渡します。しかし、検証メソッドは、newNameが参照するオブジェクトを置き換えるため、newNameオブジェクトが明示的に解放されると、検証済みのオブジェクト

が代わりに解放されます。このことから2つの問題が生じます。検証済みのnameの値に後でアクセスしようとする、setName:が保持しているオブジェクトが解放されているためクラッシュが発生し、validateName:error:メソッドに渡されていたnameオブジェクトがメモリリークとなります。

リスト 4 validateName:error:の誤った呼び出し

```
NSString *newName = [[NSString alloc] initWithString:@"freddy"];
NSError *outError = nil;

if ([person validateName:&newName error:&outError]) {
    // 値を設定。この値はnewNameオブジェクトを保持
    [person setName:newName];
}
else {
    // 値が無効であることをユーザに通知
}
[newName release];
```

リスト 5の例は、validateName:error:メソッドを呼び出す前にnewNameオブジェクトを自動解放することによって、これらの問題を回避します。元のオブジェクトは自動解放によって解放され、検証済みオブジェクトはsetValue:forKey:メッセージの結果としてレシーバに保持されます。

リスト 5 validateName:error:の正しい呼び出し

```
NSString *newName = [[[NSString alloc] initWithString:@"freddy"] autorelease];
NSError *outError = nil;

if ([person validateName:&newName error:&outError]) {
    // 値を設定。この値はnewNameオブジェクトを保持
    [person setName:newName];
}
else {
    // 値が無効であることをユーザに通知
}
```

KVOの準拠

特定のプロパティについてクラスをKVC互換にするには、そのプロパティについて`valueForKey:`および`setValue:forKey:`が機能するために必要なメソッドを、そのクラスに実装しなければなりません。

属性および対1リレーションシップのKVC互換性

プロパティが属性または対1リレーションシップの場合、KVC互換にするには次の要件を満たす必要があります。

- クラスが、`-<key>`または`-is<Key>`という名前のメソッドを実装しているか、`<key>`または`_<key>`というインスタンス変数を持っている。

多くの場合キー名は小文字で始まりますが、KVCはURLのように大文字で始まるキー名も扱えるようになっています。

- プロパティが可変の場合は、`-set<Key>`も実装する。
- `-set<Key>`メソッドの実装では、検証を実行しない。
- 検証の実行がキーに対して適切な場合は、クラスに`-validate<Key>:error:`を実装する。

インデックス付き対多リレーションシップのKVC互換性

インデックス付き対多リレーションシップの場合、KVC互換にするには次の要件を満たす必要があります。

- 配列を返す`-<key>`という名前のメソッドを実装する。
- あるいは、`<key>`または`_<key>`という名前の配列インスタンス変数を持っている。
- あるいは、`-countOf<Key>`メソッドおよび、`-objectIn<Key>AtIndex:`または`-<key>AtIndexes:`のいずれかのメソッドを実装する。
- 必要であれば、`-get<Key>:range:`も実装してパフォーマンスを改善できる。

可変のインデックス付き対多リレーションシップの場合、KVC互換にするにはさらに次の要件を満たす必要があります。

- `-insertObject:in<Key>AtIndex:`および`-insert<Key>:atIndexes:`の一方または両方を実装する。

- `-removeObjectFrom<Key>AtIndex:` および `-remove<Key>AtIndexes:` の一方または両方を実装する。
- 必要であれば、`-replaceObjectIn<Key>AtIndex:withObject:` または `-replace<Key>AtIndexes:with<Key>:` も実装してパフォーマンスを改善できる。

順序なし対多リレーションシップのKVC互換性

順序なし対多リレーションシップの場合、KVC互換にするには次の要件を満たす必要があります。

- 集合を返す `-<key>` という名前のメソッドを実装する。
- あるいは、`<key>` または `_<key>` という名前の集合インスタンス変数を持っている。
- あるいは、`-countOf<Key>`、`-enumeratorOf<Key>`、`-memberOf<Key>`: の各メソッドを実装する。

可変の順序なし対多リレーションシップの場合、KVC互換にするにはさらに次の要件を満たす必要があります。

- `-add<Key>Object:` および `-add<Key>:` の一方または両方を実装する。
- `-remove<Key>Object:` および `-remove<Key>:` の一方または両方を実装する。
- 必要であれば、`-intersect<Key>:` および `-set<Key>:` も実装してパフォーマンスを改善できる。

スカラおよび構造体のサポート

キー値コーディングでは、NSNumberおよびNSValueのインスタンスで値を自動的にラップしたり、またはそのラップを解除することによって、スカラ値とデータ構造体をサポートします。

非オブジェクト値の表現

valueForKey:およびsetValue:forKey:のデフォルトの実装では、非オブジェクトデータ型であるスカラや構造体について、オブジェクトの自動ラップをサポートします。

valueForKey:は、指定されたキーの値を供給するアクセサメソッドまたはインスタンス変数を確定すると、戻り型とデータ型を調べます。返される値がオブジェクトでない場合、その値についてNSNumberオブジェクトまたはNSValueオブジェクトが作成され、値の代わりに返されます。

同様に、setValue:forKey:は、指定されたキーについて、対応するアクセサまたはインスタンス変数が要求するデータ型を確認します。データ型がオブジェクトでない場合は、適切な<type>Valueメソッドを使用して、渡されたオブジェクトから値が抽出されます。

nil値の処理

setValue:forKey:が呼び出され、オブジェクトでないプロパティの値としてnilが渡されると、また別の問題が生じます。一般化されている適切なアクションはありません。オブジェクトでないプロパティの値としてnilが渡されると、レシーバにはsetNilValueForKey:メッセージが送信されます。setNilValueForKey:のデフォルトの実装では、NSInvalidArgumentExceptionの例外が発生します。サブクラスでこのメソッドをオーバーライドして、実装固有の適切な動作をさせることができます。

リスト 1のコード例は、人の年齢（浮動小数点値）をnilに設定する試みに対して0に設定することです。

注意： レシーバのクラスにおいてNSObjectのunableToSetNilForKey:の実装をオーバーライドしている場合には、バイナリレベルの下位互換性確保のために、setNilValueForKey:の代わりにunableToSetNilForKey:が呼び出されます。

リスト 1 setNilValueForKey:の実装例

```
- (void)setNilValueForKey:(NSString *)theKey
{
    if ([theKey isEqualToString:@"age"]) {
        [self setValue:[NSNumber numberWithFloat:0.0] forKey:@"age"];
    } else
        [super setNilValueForKey:theKey];
}
```

}

スカラ型のラップとその解除

表 1に、NSNumberインスタンスでラップされるスカラ型を列挙します。

表 1 NSNumberオブジェクトにおいてラップされるスカラ型

データ型	作成メソッド	アクセサメソッド
BOOL	numberWithBool:	boolValue
char	numberWithChar:	charValue
double	numberWithDouble:	doubleValue
float	numberWithFloat:	floatValue
int	numberWithInt:	intValue
long	numberWithLong:	longValue
long long	numberWithLongLong:	longLongValue
short	numberWithShort:	shortValue
unsigned char	numberWithUnsignedChar:	unsignedChar
unsigned int	numberWithUnsignedInt:	unsignedInt
unsigned long	numberWithUnsignedLong:	unsignedLong
unsigned long long	numberWithUnsignedLongLong:	unsignedLongLong
unsigned short	numberWithUnsignedShort:	unsignedShort

構造体のラップとその解除

表 2（34 ページ）に、一般的な構造体であるNSPoint、NSRange、NSRect、NSSizeをラップするために用いる、作成メソッド、アクセサメソッドを列挙します。

表 2 NSValueを使用してラップされる一般的な構造体

データ型	作成メソッド	アクセサメソッド
NSPoint	valueWithPoint:	pointValue
NSRange	valueWithRange:	rangeValue

データ型	作成メソッド	アクセサメソッド
NSRect	valueWithRect: (Mac OS Xのみ)	rectValue
NSSize	valueWithSize:	sizeValue

自動ラップや自動ラップ解除は、NSPoint、NSRange、NSRect、NSSizeに限定されるわけではありません。構造体型（Objective-Cの型エンコーディング文字列が「{」で始まる型）は、NSValueオブジェクトにラップ可能です。たとえば次のような構造体とクラスがあったとします。

```
typedef struct {
    float x, y, z;
} ThreeFloats;

@interface MyClass
- (void)setThreeFloats:(ThreeFloats)threeFloats;
- (ThreeFloats)threeFloats;
@end
```

MyClassのインスタンスに、@"threeFloats"をパラメータとしてメッセージvalueForKey:を送信すると、MyClassのthreeFloatsメソッドが起動され、結果はNSValueにラップして返されます。同様に、MyClassのインスタンスに、ThreeFloatsをラップしたNSValueオブジェクトをパラメータとしてsetValue:forKey:メッセージを送信すると、NSValueオブジェクトにgetValue:メッセージを送信して得られた結果をパラメータとして、setThreeFloats:が起動されます。

注意： この仕組みは参照カウントやガベージコレクションを考慮していないので、オブジェクトポインタを収容する構造体型を扱う場合は注意が必要です。

コレクション演算子

コレクション演算子を使うと、コレクションに属する項目に対するアクションを、キーパス記法とアクション演算子で記述できます。この章では、使用可能なコレクション演算子について説明し、キーパスの例、実行結果を示します。

コレクション演算子はキーパスの特別な形態で、パラメータとしてvalueForKeyPath:メソッドに渡して使います。この演算子は先頭に「@」がついていることで識別できます。コレクション演算子の左にキーパスを指定した場合、そのキーパスは演算で使用する配列または集合の特定に使用されます。配列や集合はレシーバを起点として相対的に決まります。コレクション演算子の右にあるキーパスは、演算で使用するコレクション内の項目のプロパティを指定します。図1に演算子とキーパスの書式を例示します。

図1 演算子とキーパスの書式

`keypathToCollection.@collectionOperator.keypathToProperty`

Left key path Collection operator Right key path

@countを除き、コレクション演算子はすべて、演算子の右側にキーパスを指定する必要があります。

注意： 現状では、独自のコレクション演算子を定義することはできません。

返されるオブジェクト値の型は演算子によって異なります。

- 単純型コレクション演算子は、右側に指定したキーに応じて、文字列、数値、日付のいずれかを返します（「[単純型コレクション演算子](#)」（38 ページ）を参照）。
- オブジェクト演算子はNSArrayのインスタンスを返します。これは一般のオブジェクト演算子と同様です（「[オブジェクト演算子](#)」（40 ページ）を参照）。
- 配列演算子、集合演算子は、演算子に応じ、配列オブジェクトまたは集合オブジェクトを返します（「[配列演算子、集合演算子](#)」（41 ページ）を参照）。

例に用いるデータ

「[単純型コレクション演算子](#)」（38 ページ）、「[オブジェクト演算子](#)」（40 ページ）、「[配列演算子、集合演算子](#)」（41 ページ）では、各演算子の説明に続いて、当該演算子を使ったキーパスの記述例とその結果を示しています。例示のために仮想的なクラスが必要なので、ここではTransactionクラスを使います。Transactionクラスには、小切手用紙を模した簡単な書式がカプセル化されています。ここには、payee (NSString型)、amount (NSNumber型)、date (NSDate型) という3つのプロパティがあります。

Transactionのインスタンスはコレクションオブジェクトに収容されます。ここでは仮にこれを、NSArrayのインスタンスである、transactionsという変数で表します。NSSetコレクションに作用する演算子については、transactionsコレクションがNSSetのインスタンスであると想定してください。これについては、該当する演算子の項で説明します。

transactionsコレクションに対して演算子を施した結果を示すために、例として次のようなデータを使います。表1の各行は、Transactionの各インスタンスに格納されたデータを表します。表の各値は、見やすいように書式を整えてあります。演算子を施した結果についても、同様に書式を整えて示します。

表1 Transactionsオブジェクトのデータ例

payeeの値	amountの値（金額として整形）	dateの値（月、日、年の順に整形）
Green Power	\$120.00	Dec 1, 2009
Green Power	\$150.00	Jan 1, 2010
Green Power	\$170.00	Feb 1, 2010
Car Loan	\$250.00	Jan 15, 2010
Car Loan	\$250.00	Feb 15, 2010
Car Loan	\$250.00	Mar 15, 2010
General Cable	\$120.00	Dec 1, 2009
General Cable	\$155.00	Jan 1, 2010
General Cable	\$120.00	Feb 1, 2010
Mortgage	\$1,250.00	Jan 15, 2010
Mortgage	\$1,250.00	Feb 15, 2010
Mortgage	\$1,250.00	Mar 15, 2010
Animal Hospital	\$600.00	Jul 15, 2010

単純型コレクション演算子

単純型コレクション演算子は、配列または集合を対象とし、演算子の右側に記述したプロパティに対して作用します。

@avg

@avg演算子はvalueForKeyPath:を使用して、演算子の右側のキーパスにより指定されたプロパティの値を取得し、その値をdoubleに変換した上で、平均値をNSNumberのインスタンスとして返します。値がnilであれば、0として扱います。

次の例は、transactionsの各オブジェクトについて、金額（amount）の平均値を返します。

```
NSNumber *transactionAverage=[transactions valueForKeyPath:@"@avg.amount"];
```

transactionAverageの結果を整形して表すと「\$456.54」となります。

@count

@count演算子は、左側のキーパスで指定したコレクションに含まれるオブジェクトの個数を、NSNumberのインスタンスとして返します。演算子の右側のキーパスは無視します。

次の例は、transactionsに含まれるTransactionオブジェクトの個数を返します。

```
NSNumber *numberOfTransactions=[transactions valueForKeyPath:@"@count"];
```

numberOfTransactionsの値は13となります。

@max

@max演算子は、演算子の右側のキーパスで指定されたプロパティの値を比較し、最大値を返します。最大値の判定には、指定されたキーパスに対応するオブジェクトのcompare:メソッドを使います。比較対象のプロパティオブジェクトは、互いの比較が可能でなければなりません。キーパスの右側の値がnilであれば、無視します。

次の例は、transactionsに含まれるTransactionオブジェクトのdateの最大値（最も新しい取引日）を返します。

```
NSDate *latestDate=[transactions valueForKeyPath:@"@max.date"];
```

latestDateの値を整形して表すと「Jul 15, 2010」となります。

amountSumの結果を整形して表すと「\$5,935.00」となります。

@min

@min演算子は、演算子の右側のキーパスで指定されたプロパティの値を比較し、最小値を返します。最小値の判定には、指定されたキーパスに対応するオブジェクトのcompare:メソッドを使います。比較対象のプロパティオブジェクトは、互いの比較が可能でなければなりません。キーパスの右側の値がnilであれば、無視します。

次の例は、transactionsに含まれるTransactionオブジェクトのdateの最小値（最も古い取引日）を返します。

```
NSDate *earliestDate=[transactions valueForKeyPath:@"@min.date"];
```

earliestDateの値を整形して表すと「Dec 1, 2009」となります。

@sum

@sum演算子は、その右側のキーパスで指定されたプロパティ値の合計を返します。各値をdoubleに変換して合計し、NSNumberのインスタンスとしてラップして返します。キーパスの右側の値がnilであれば、無視します。

次の例は、transactionsの各取引についてamountsプロパティの合計値を返します。

```
NSNumber *amountSum=[transactions valueForKeyPath:@"@sum.amount"];
```

amountSumの結果を整形して表すと「\$5,935.00」となります。

オブジェクト演算子

オブジェクト演算子は、単一のコレクションに適用した結果を返します。

@distinctUnionOfObjects

@distinctUnionOfObjects演算子は、その右側のキーパスで指定されたプロパティの値を、重複を除いて、配列の形で返します。

次の例は、transactionsに含まれる各取引からpayeeプロパティの値を取り出して重複を除き、配列として返します。

```
NSArray *payees=[transactions valueForKeyPath:@"@distinctUnionOfObjects.payee"];
```

結果として得られるpayees配列は、Car Loan、General Cable、Animal Hospital、Green Power、Mortgageの各文字列から成ります。

@unionOfObjects演算子も同様ですが、重複を除かない点が異なります。

重要： コレクションにnilが含まれていると例外が発生します。

@unionOfObjects

@unionOfObjects演算子は、その右側のキーパスで指定されたプロパティの値を、重複を除かずに、配列の形で返します。「@distinctUnionOfObjects」と違い、重複があってもそのままです。

次の例は、transactionsに含まれる各取引からpayeeプロパティの値を取り出し、配列として返します。

```
NSArray *payees=[transactions valueForKeyPath:@"@unionOfObjects.payee"];
```

結果として得られるpayees配列は、Green Power、Green Power、Green Power、Car Loan、Car Loan、Car Loan、General Cable、General Cable、General Cable、Mortgage、Mortgage、Mortgage、Animal Hospitalの各文字列から成ります。

@distinctUnionOfArrays演算子も同様ですが、重複を除く点が異なります。

重要： コレクションにnilが含まれていると例外が発生します。

配列演算子、集合演算子

配列演算子や集合演算子は、入れ子になったコレクション、すなわち、各項目がコレクションであるようなコレクションに作用します。

変数arrayOfTransactionsを各演算子の例に使います。これはTransactionオブジェクトの配列2つから成る配列です。

入れ子になったコレクションを作成するコード例を示します。

```
// 配列を要素として追加できる配列を作成
self.arrayOfTransactionsArray = [NSMutableArray array];

// 先の例で使ったオブジェクトの配列を追加
[arrayOfTransactionsArray addObject:transactions];

// 値が異なる、別のオブジェクトの配列を追加
[arrayOfTransactionsArrays addObject:moreTransactions];
```

第1のTransactionオブジェクト配列には表 1 (38 ページ) のデータ、第2の配列 (moreTransactions) には表 2に示すTransactionオブジェクトが含まれるとします。

表 2 配列moreTransactionsに含まれるTransactionのデータ

payeeの値	amountの値 (金額として整形)	dateの値 (月、日、年の順に整形)
General Cable - Cottage	\$120.00	Dec 18, 2009
General Cable - Cottage	\$155.00	Jan 9, 2010
General Cable - Cottage	\$120.00	Dec 1, 2010
Second Mortgage	\$1,250.00	Nov 15, 2010
Second Mortgage	\$1,250.00	Sep 20, 2010
Second Mortgage	\$1,250.00	Feb 12, 2010
Hobby Shop	\$600.00	Jun 14, 2010

@distinctUnionOfArrays

@distinctUnionOfArrays演算子は、その右側のキーパスで指定されたプロパティの値を、重複を除いて、配列の形で返します。

次の例は、arrayOfTransactionsArraysの各配列に含まれる各取引からpayeeプロパティの値を取り出して重複を除き、配列として返します。

```
NSArray *payees=[arrayOfTransactionsArrays  
valueForKeyPath:@"distinctUnionOfArrays.payee"];
```

結果として得られるpayees配列は、Hobby Shop、Mortgage、Animal Hospital、Second Mortgage、Car Loan、General Cable - Cottage、General Cable、Green Powerの各文字列から成ります。

@unionOfArrays演算子も同様ですが、重複を除かない点が異なります。

重要： コレクションにnilが含まれていると例外が発生します。

@unionOfArrays

@unionOfArrays演算子は、その右側のキープスで指定されたプロパティの値を、重複を除かずに、配列の形で返します。@distinctUnionOfArraysと違い、重複があってもそのままです。

次の例は、arrayOfTransactionsArraysの各配列に含まれる各取引からpayeeプロパティの値を取り出し、配列として返します。

```
NSArray *payees=[arrayOfTransactionsArrays  
valueForKeyPath:@"unionOfArrays.payee"];
```

結果として得られるpayees配列は、Green Power、Green Power、Green Power、Car Loan、Car Loan、Car Loan、General Cable、General Cable、General Cable、Mortgage、Mortgage、Mortgage、Animal Hospital、General Cable - Cottage、General Cable - Cottage、General Cable - Cottage、Second Mortgage、Second Mortgage、Second Mortgage、Hobby Shopの各文字列から成ります。

重要： コレクションにnilが含まれていると例外が発生します。

@distinctUnionOfSets

@distinctUnionOfSets演算子は、その右側のキープスで指定されたプロパティの値を、重複を除いて、配列の形で返します。

@distinctUnionOfArraysと同様ですが、Transactionのインスタンスから成るNSSetを含む、NSSetのインスタンスを対象とします（配列ではなく集合）。戻り値はNSSetのインスタンスです。同じデータ集合を例にすると、戻り値の集合に含まれる文字列は、「@distinctUnionOfArrays」（41 ページ）の場合と同じです。

重要： コレクションにnilが含まれていると例外が発生します。

アクセサ検索の実装の詳細

キー値コーディングでは、値の取得や設定を行う方法として、インスタンス変数に直接アクセスする前に、まずアクセサメソッドの使用が試みられます。この記事では、キー値コーディングのメソッドがどのようにして値のアクセス方法を決定するかについて説明します。

単純型属性に対するアクセサの検索パターン

setValue:forKey:におけるデフォルトの検索パターン

プロパティについて、setValue:forKey:のデフォルトの実装が呼び出されると、次の検索パターンが使用されます。

1. set<Key>:というパターンに名前が一致するアクセサメソッドがないかどうか、レシーバのクラスが検索されます。
2. アクセサメソッドが見つからず、レシーバのクラスメソッドである accessInstanceVariablesDirectlyがYESを返した場合、_<key>、_is<Key>、<key>、そして is<Key>の順番で、それぞれのパターンに名前が一致するインスタンス変数がないかどうか、レシーバが検索されます。
3. 一致するアクセサまたはインスタンス変数が見つかった場合は、それを使用して値が設定されます。必要な場合は、「[非オブジェクト値の表現](#)」（33 ページ）で説明しているように、オブジェクトから値が抽出されます。
4. 該当するアクセサまたはインスタンス変数が見つからない場合は、レシーバに対して setValue:forUndefinedKey:が呼び出されます。

valueForKey:におけるデフォルトの検索パターン

レシーバを対象にvalueForKey:のデフォルトの実装が呼び出されると、次の検索パターンが使用されます。

1. get<Key>、<key>、そして is<Key>の順で、いずれかのパターンに名前が一致するアクセサメソッドがないかどうか、レシーバのクラスが検索されます。見つかった場合はそのメソッドを起動します。メソッドの戻り値型がオブジェクトポインタであれば、結果をそのまま返します。NSNumberに変換できるスカラー型であれば、NSNumberに変換して返します。そうでなければ NSValueに変換して返します。Mac OS X v10.5以降、NSPoint、NSRange、NSRect、NSSizeに限らずどのような型でも、NSValueオブジェクトに変換するようになりました。

2. そうでない（単純アクセサメソッドが見つからなかった）場合は、`countOf<Key>`、`objectIn<Key>AtIndex:`（NSArrayクラスに定義されたプリミティブなメソッドに相当）、`<key>AtIndexes:`（NSArrayの`objectsAtIndexes:`メソッドに相当）というパターンのメソッドが、レシーバクラスに定義されていないかどうか調べます。

`countOf<Key>`メソッドと、それ以外のメソッドのうちいずれか一方が見つければ、NSArrayのあらゆるメソッドに応答するコレクションプロキシオブジェクトを返します。コレクションプロキシオブジェクトにNSArrayのメッセージを送信すると、`countOf<Key>`、`objectIn<Key>AtIndex:`、`<key>AtIndexes:`の各メッセージを適切に組み合わせたものが、本来の`valueForKey:`のレシーバに送信されます。レシーバのクラスに、`get<Key>:range:`というメソッドが実装されていれば、性能向上のため、このメソッドを使うようになっています。

3. そうでない（単純アクセサメソッドも配列アクセスメソッドの組も見つからなかった）場合は、`countOf<Key>`、`enumeratorOf<Key>`、`memberOf<Key>`（NSSetクラスに定義されたプリミティブなメソッドに相当）というパターンのメソッドが3つとも、レシーバクラスに定義されていないかどうか調べます。

3つとも見つかった場合は、NSSetのあらゆるメソッドに応答するコレクションプロキシオブジェクトを返します。コレクションプロキシオブジェクトにNSSetのメッセージを送信すると、`countOf<Key>`、`enumeratorOf<Key>`、`memberOf<Key>`の各メッセージを組み合わせたものが、本来の`valueForKey:`のレシーバに送信されます。

4. そうでない（単純アクセサメソッドもコレクションアクセスメソッドの組も見つからなかった）場合、レシーバのクラスメソッド`accessInstanceVariablesDirectly`がYESを返せば、当該レシーバクラスに、名前が`_<key>`、`_is<Key>`、`<key>`、`is<Key>`のいずれかに合致するインスタンス変数がないか、この順序で検索します。見つかった場合は、レシーバの当該インスタンス変数の値を返します。NSNumberに変換できるスカラ型であれば、NSNumberに変換して返します。そうでなければNSValueに変換して返します。Mac OS X v10.5以降、NSPoint、NSRange、NSRect、NSSizeに限らずどのような型でも、NSValueオブジェクトに変換するようになりました。
5. いずれも見つからなかった場合は、`valueForUndefinedKey:`を起動し、その結果を返します（これがデフォルトの実装）。

順序付きコレクションに対するアクセサの検索パターン

`mutableArrayValueForKey:`のデフォルトの検索パターンを以下に示します。

1. レシーバのクラスに、`insertObject:in<Key>AtIndex:`および`removeObjectFrom<Key>AtIndex:`（NSMutableArrayのプリミティブメソッド`insertObject:atIndex:`および`removeObjectAtIndex:`に相当）、または`insert<Key>:atIndexes:`および`remove<Key>AtIndexes:`（NSMutableArrayの`insertObjects:atIndexes:`および`removeObjectsAtIndexes:`に相当）に合致するメソッドの組があるかどうか調べます。

いずれかの挿入メソッド、いずれかの削除メソッドが見つかった場合、コレクションプロキシオブジェクトにNSMutableArrayのメッセージを送信すると、`insertObject:in<Key>AtIndex:`、`removeObjectFrom<Key>AtIndex:`、`insert<Key>:atIndexes:`、`remove<Key>AtIndexes:`を適切に組み合わせたものが、本来の`mutableArrayValueForKey:`のレシーバに送信されます。

レシーバのクラスに`replaceObjectIn<Key>AtIndex:withObject:`または`replace<Key>AtIndexes:with<Key>:`という置換メソッドが実装されていれば、性能向上のため、このメソッドを使うようになっています。

2. そうでない場合は、`set<Key>:`というパターンのアクセサメソッドが、レシーバのクラスに定義されていないかどうか調べます。見つかった場合は、コレクションプロキシオブジェクトに`NSMutableArray`メッセージを送信すると、`set<Key>:`メッセージが、本来の`mutableArrayValueForKey:`のレシーバに送信されます。前ステップで述べた、インデックス付きアクセサメソッドを実装すると、より効率的になります。
3. そうでない場合、レシーバのクラスメソッド`accessInstanceVariablesDirectly`がYESを返せば、当該レシーバクラスに、名前が`_key<`、`<key>`のいずれかに合致するインスタンス変数がないか、この順序で検索します。

見つかった場合、コレクションプロキシオブジェクトに`NSMutableArray`のメッセージを送信すると、当該インスタンス変数の値（一般に`NSMutableArray`またはそのサブクラスのインスタンス）に転送されます。

4. そうでない場合、プロキシが`NSMutableArray`メッセージを受信する都度、`setValue:forUndefinedKey:`メッセージを本来の`mutableArrayValueForKey:`メッセージのレシーバに送信して得られる、可変コレクションプロキシオブジェクトを返します。

`setValue:forUndefinedKey:`のデフォルトの実装は、例外`NSUndefinedKeyException`が発生するようになっていますが、オーバーライドしてこの動作を変えても構いません。

注意： ステップ2に説明した、`set<Key>:`メッセージを繰り返し送信すると、性能に問題が生じる恐れがあります。これを回避するため、キー値コーディング互換クラスには、ステップ1の条件を満たすメソッドを実装してください。

一意順序付きコレクションに対するアクセサの検索パターン

`mutableOrderedSetValueForKey:`のデフォルトの実装は、`valueForKey`と同じ、単純アクセサメソッドおよび順序付き集合アクセサメソッド（「[valueForKey:におけるデフォルトの検索パターン](#)」（43 ページ）を参照）を認識し、インスタンス変数に直接アクセスする方針についても同様となっていますが、`valueForKey:`が返す不変コレクションではなく、可変コレクションプロキシオブジェクトを返す点が異なります。デフォルトの検索パターンは次のようになります。

1. レシーバのクラスに、`insertObject:in<Key>AtIndex:`および`removeObjectFrom<Key>AtIndex:`（いずれも`NSMutableArrayOrderedSet`プリミティブメソッドに相当）と、`insert<Key>:atIndexes:`および`remove<Key>atIndexes:`（`insertObjects:atIndexes:`および`removeObjectsAtIndexes:`に相当）に合致するメソッドがないかどうか調べます。

いずれかの挿入メソッド、いずれかの削除メソッドが見つかった場合、コレクションプロキシオブジェクトに`NSMutableArrayOrderedSet`のメッセージを送信すると、`insertObject:in<Key>AtIndex:`、`removeObjectFrom<Key>AtIndex:`、`insert<Key>:atIndexes:`、`remove<Key>atIndexes:`を適切に組み合わせたものが、本来の`mutableOrderedSetValueForKey:`のレシーバに送信されます。

レシーバのクラスに`replaceObjectIn<Key>AtIndex:withObject:`または`replace<Key>AtIndexes:with<Key>:`という置換メソッドが実装されていれば、性能向上のため、このメソッドを使うようになっています。

2. そうでない場合は、`set<Key>:`というパターンのアクセサメソッドが、レシーバのクラスに定義されていないかどうか調べます。見つかった場合は、コレクションプロキシオブジェクトの`NSMutableOrderedSet`メッセージを送信すると、`set<Key>:`メッセージが、本来の`mutableOrderedSetValueForKey:`のレシーバに送信されます。
3. そうでない場合、レシーバのクラスメソッド`accessInstanceVariablesDirectly`がYESを返せば、当該レシーバクラスに、名前が`_<key>`、`<key>`のいずれかに合致するインスタンス変数がないか、この順序で検索します。見つかった場合、コレクションプロキシオブジェクトに`NSMutableOrderedSet`のメッセージを送信すると、当該インスタンス変数の値（一般に`NSMutableOrderedSet`またはそのサブクラスのインスタンス）に転送されます。
4. そうでない場合は可変コレクションプロキシオブジェクトを返します。コレクションプロキシオブジェクトに`NSMutableOrderedSet`メッセージを送信すると、`setValue:forUndefinedKey:`メッセージが、本来の`mutableOrderedSetValueForKey:`のレシーバに送信されます。`setValue:forUndefinedKey:`のデフォルトの実装は、例外`NSUndefinedKeyException`が発生するようになっていますが、オーバーライドしてこの動作を変えても構いません。

注意： ステップ2に説明した、`set<Key>:`メッセージを繰り返し送信すると、性能に問題が生じる恐れがあります。これを回避するため、キー値コーディング互換クラスには、ステップ1の条件を満たす挿入および削除メソッドを実装してください。さらに置換メソッドも実装するとよいでしょう。

順序なしコレクションに対するアクセサの検索パターン

`mutableSetValueForKey:`のデフォルトの検索パターンを以下に示します。

1. レシーバのクラスに、`add<Key>Object:`および`remove<Key>Object:`（`NSMutableSet`のプリミティブメソッド`addObject:`および`removeObject:`に相当）と、`add<Key>:`および`remove<Key>:`（`NSMutableSet`の`unionSet:`および`minusSet:`に相当）に合致するメソッドの組があるかどうか調べます。いずれかの挿入メソッド、いずれかの削除メソッドが見つかった場合、コレクションプロキシオブジェクトに`NSMutableSet`のメッセージを送信すると、`add<Key>Object:`、`remove<Key>Object:`、`add<Key>:`、`remove<Key>:`を適切に組み合わせたものが、本来の`mutableSetValueForKey:`のレシーバに送信されます。

レシーバのクラスに`intersect<Key>:`または`set<Key>:`というメソッドが実装されていれば、性能向上のため、このメソッドを使うようになっています。

2. レシーバが管理オブジェクトであれば、検索パターンはここで終わりです（以下は非管理オブジェクトの場合に適用）。詳しくは『*Model Object Implementation Guide*』の「**Managed Object Accessor Methods**」を参照してください。
3. そうでない場合は、`set<Key>:`というパターンのアクセサメソッドが、レシーバのクラスに定義されていないかどうか調べます。見つかった場合は、コレクションプロキシオブジェクトに`NSMutableSet`メッセージを送信すると、`set<Key>:`メッセージが、本来の`mutableSetValueForKey:`のレシーバに送信されます。

4. そうでない場合、レシーバのクラスメソッド`accessInstanceVariablesDirectly`がYESを返せば、当該レシーバクラスに、名前が`_<key>`、`<key>`のいずれかに合致するインスタンス変数がないか、この順序で検索します。見つかった場合、コレクションプロキシオブジェクトに`NSMutableDictionary`のメッセージを送信すると、当該インスタンス変数の値（一般に`NSMutableDictionary`またはそのサブクラスのインスタンス）に転送されます。
5. そうでない場合は可変コレクションプロキシオブジェクトを返します。コレクションプロキシオブジェクトに`NSMutableDictionary`のメッセージを送信すると、`setValue:forUndefinedKey:`メッセージが、本来の`mutableSetValueForKey:`のレシーバに送信されます。

注意： ステップ3に説明した、`set<Key>`:メッセージを繰り返し送信すると、性能に問題が生じる恐れがあります。これを回避するため、キー値コーディング互換クラスには、ステップ1の条件を満たすメソッドを実装してください。

プロパティのリレーションシップの記述

クラスの記述では、クラス内で対1リレーションシップや対多リレーションシップにあるプロパティを記述することができます。クラスのプロパティどうしのリレーションシップを定義することで、これらプロパティをキー値コーディングで使用する場合に、より高度で柔軟性の高い処理が可能となります。

クラスの記述

`NSClassDescription`は、クラスのメタデータを取得するインターフェースを備えた基本クラスです。クラス記述オブジェクトは、特定クラスのオブジェクトについて利用可能な属性を記録するとともに、そのクラスのオブジェクトとほかのオブジェクトとの関係（1対1、1対多、および逆向き）を記録します。たとえば、`attributes`メソッドは、クラスについて定義されているすべての属性のリストを返します。`toManyRelationshipKeys`メソッドおよび`toOneRelationshipKeys`メソッドは、対多リレーションシップおよび対1リレーションシップを定義するキーの配列を返します。また、`inverseRelationshipKey:`メソッドは、指定したキーに対応するリレーションシップのデスティネーションを起点としてレシーバに向かう逆方向のリレーションシップの名前を返します。

`NSClassDescription`は、リレーションシップを定義するメソッドは定義しません。具象サブクラスがこれらのメソッドを定義する必要があります。メソッドを作成したら、クラスの記述を `NSClassDescription` の `registerClassDescription:forClass:` クラスメソッドに登録します。

`NSScriptClassDescription`は、Cocoaで提供されている、`NSClassDescription`の唯一の具象サブクラスです。このサブクラスは、アプリケーションのスクリプティング情報をカプセル化します。

パフォーマンスに関する考慮事項

キー値コーディングは効率的である反面、間接的な段階が間に1つ入るため、直接メソッドを呼び出す場合と比べて、わずかながら処理に時間がかかるようになります。キー値コーディングは、その柔軟性がメリットとなる場合にのみ使用するべきです。

将来、処理のパフォーマンスが最適化される可能性はありますが、キー値コーディングに準拠するための基本的な方法が変わるわけではありません。

キー値コーディングメソッドのオーバーライド

`valueForKey:`をはじめとするキー値コーディングメソッドのデフォルトの実装では、効率向上のためにObjective-Cランタイム情報がキャッシュに格納されます。これらデフォルトの実装をオーバーライドする場合は、アプリケーションのパフォーマンスに悪影響を与えないよう十分注意する必要があります。

対多リレーションシップの最適化

アクセサのインデックス付きの形式を使用して実装された対多リレーションシップは、多くの場合、パフォーマンス上の大きなメリットが得られます。

対多コレクションに対しては少なくとも、インデックス付きアクセサを実装するよう推奨します。詳しくは「[対多プロパティに対するコレクション型アクセサの命名パターン](#)」（20 ページ）を参照してください。

書類の改訂履歴

この表は「キー値コーディングプログラミングガイド」の改訂履歴です。

日付	メモ
2011-06-06	Mac OS X 10.7用に更新し、一意化の順序付きリレーションシップを含めました。
2010-09-01	valueWithRect:はMac OS X専用であることを明記しました。「コレクション演算子」の章を改訂しました。
2010-04-28	誤字を訂正しました。
2010-01-20	「依存キーの登録」とCore Data管理オブジェクトの相互関係を明確にしました。「順序なしリレーションシップの可変アクセサ」の説明を訂正しました。
2009-02-04	タスク情報とサンプルコードを大幅に追加しました。
2007-06-06	配列演算子および集合演算子がnil値を生じるという警告を追加しました。
2007-01-08	validateName:error:のエラーパラメータを検査し、有効なNSErrorオブジェクトが返ることを確認する旨の注意を追加しました。
2006-06-28	『キー値監視プログラミングガイド』を関連文書の一覧に追加しました。
2006-04-04	-get<Key>:range:アクセサパターンのメソッドシグネチャの例を追加しました。
2006-03-08	@distinctUnionOfArraysの例で変数名を訂正しました。
2005-08-11	「Why Use Key-Value Coding」のタイトルを変更しました。
2005-07-07	コレクション演算子@unionOfSetsおよび@distinctUnionOfSetsの説明を追加しました。validateValue:forKey:の戻り値の条件を明確にしました。
2005-04-29	細かな誤字を訂正しました。
2004-08-31	目次を更新しました。
	「 コレクション演算子 」（37 ページ）において、@sum配列演算子がNSNumberを返すことを明記しました。
	細かな誤字を訂正しました。
2004-06-28	誤字を訂正しました。

日付	メモ
2004-04-19	「 対多プロパティに対するコレクション型アクセサの命名パターン 」（20 ページ）において、インデックス付きアクセサメソッドでプロパティが配列として表現されることを明記しました。
2003-10-15	Mac OS X 10.3向けに『 <i>Key-Value Coding</i> 』の内容を改訂しました。
2003-07-19	Mac OS X 10.3で廃止されたメソッドに関して、「 アクセサ検索の実装の詳細 」（43 ページ）の記述を更新しました。
2002-11-12	『 <i>Key-Value Coding</i> 』に改訂履歴を追加しました。