プレディケート プログラミングガイド

Developer

目次

はじめに

プレディケート(Predicate:叙述)は、Cocoaでクエリ(検索条件)を指定する一般的な手段です。プレディケートシステムは、Core DataやSpotlightなどの広い範囲で使用できます。この文書では、プレディケートの概要、使用方法、構文、および制限事項について説明します。

はじめに

Cocoaにおける**プレディケート**は、ブール値(真または偽)に評価される論理文です。プレディケートには、*比較*と*複合*の2種類があります。

- **比較プレディケート**は、**演算子**を使用して2つの**式**を比較します。これらの式は、演算子を間に挟んでプレディケートの左辺および右辺と呼ばれます。比較プレディケートは、式の評価結果を使用して演算子を呼び出した結果を返します。
- **複合プレディケート**は、2つ以上のプレディケートの評価結果を比較するか、別のプレディケートを否定します。

Cocoaでは、次のようなさまざまなタイプのプレディケートがサポートされています。

- 単純比較(`grade == 7`や`firstName like 'Mark'`など)
- 大文字と小文字または発音区別符号を区別しない照合(`name contains[cd] 'citroen'`など)
- 論理演算(`(firstName beginswith 'M') AND (lastName like 'Adderley')`など)

リレーションシップ(`group.name matches 'work.*'`、`ALL children.age > 12`、`ANY children.age > 12`など)や演算(`@sum.items.price < 1000`など)を扱うプレディケートも作成できます。

Cocoaのプレディケートを利用すると、検索対象データの保持に使用されているストアに依存しないようにクエリをコード化できます。プレディケートを使用して、SpotlightやCore Dataによって取得されるオブジェクトの集合を制限する論理条件や、メモリ内でオブジェクトにフィルタリングを適用する論理条件を表すことができます。

プレディケートは任意のクラスのオブジェクトで使用できますが、クラスはプレディケートで使用するキーについてキー値コーディングに準拠している必要があります。

プレディケートクラス

Cocoaには、NSPredicateクラスと、そのサブクラスのNSComparisonPredicateおよびNSCompoundPredicateが用意されています。

NSPredicateクラスには、プレディケートを評価するメソッドや、文字列(firstName like 'Mark'など)からプレディケートを作成するメソッドが用意されています。文字列からプレディケートを作成するときに、NSPredicateは自動的に適切なプレディケートと式のインスタンスを作成します。状況によっては、比較プレディケートまたは複合プレディケートを自分で作成しなければならない場合もあります。このようなときは、NSComparisonPredicateクラスとNSCompoundPredicateクラスを使用します。

Cocoaでのプレディケートの式は、NSExpressionクラスのインスタンスによって表されます。最も簡単なプレディケートの式は定数値を表します。ただし多くの場合は、プレディケートで現在評価しているオブジェクトの、キーパスの値を取得する式を使用します。また、プレディケートで現在評価しているオブジェクトを表す式、変数のプレースホルダとして機能する式、または配列での演算結果を返す式を作成することもできます。

プレディケートと式の作成については、“プレディケートを作成する”（7 ページ）を参照してください。

制約(Constraint)と制限事項

プレディケートをCore DataまたはSpotlightで使用する場合は、対応するデータストアが操作対象になることに注意してください。プレディケートのクエリに特定の実装言語はありません。プレディケートのクエリは、バッキングストアに要件があれば、それに応じてSQL、XML、またはその他の形式に変換されます。

Cocoaのプレディケートシステムは、有効な範囲の演算子だけをサポートするように意図されています。使用できる演算子は、すべてのバッキングストアでサポートされているすべての演算子の和集合でも積集合でもありません。したがって、記述可能なプレディケートのクエリすべてが、すべてのバッキングストアでサポートされるとは限りません。また、すべてのバッキングストアでサポートされている演算を、NSPredicateおよびNSExpressionオブジェクトで表現できるわけでもありません。サポートされない演算子を使用しようとすると、バックエンドでプレディケートがダウングレードされたり(たとえば、大文字と小文字を区別する比較で大文字と小文字が区別されなかったり)、例外が発生したりする場合があります。たとえば次のようになります。

- matches演算子はregexを使用します。したがって、メモリ内のフィルタリングでは動作しますが、Core DataのSQLストアではサポートされません。
- Core DataのSQLストアは、クエリごとに対多の演算を1つだけサポートします。したがって、SQLストアに送信されるプレディケートには、ALL、ANY、およびINのいずれか1つの演算子(および、その演算子の1つのインスタンス)だけが含まれます。

- すべてのSQLクエリをプレディケートに変換できるわけではありません。
- `ANYKEY`演算子はSpotlightのみで使用できます。
- Spotlightではリレーションシップはサポートされません。

この文書の使い方

次の章では、Cocoaにおけるプレディケートの基本事項、プレディケートオブジェクトの作成方法と使用方法、およびプレディケート構文の定義について説明しています。

- "プレディケートを作成する"（7 ページ）では、プレディケートのインスタンスをプログラムで正しく作成する方法と、管理オブジェクトモデルからプレディケートを取得する方法を説明します。
- "プレディケートを使用する"（14 ページ）では、プレディケートの使用方法と、パフォーマンスに関するいくつかの問題について説明します。
- "NSPredicateとSpotlightのクエリ文字列の比較"（22 ページ）では、NSPredicateとSpotlightのクエリを比較します。
- "プレディケートの書式文字列の構文"（24 ページ）では、プレディケートの書式文字列の構文について説明します。
- "CocoaのプレディケートのBNF定義"（31 ページ）では、Cocoaのプレディケートをバッカスナウア記法で定義します。

プレディケートを作成する

Cocoaでプレディケートを作成する場合、書式文字列の使用、コードへの直接入力、およびプレディケートテンプレートからの作成の3つの方法があります。

書式文字列を使用してプレディケートを作成する

`NSPredicate`クラスの`predicateWithFormat…`という形式のメソッドを使用して、文字列から直接プレディケートを作成できます。プレディケートを文字列として定義し、必要な場合は変数置換を使用します。変数置換がある場合、置換は実行時に行われ、結果の文字列が解析されて対応するプレディケートと式のオブジェクトが作成されます。次の例では、2つの比較プレディケートを含む複合プレディケートを作成しています。

```
NSPredicate *predicate = [NSPredicate
    predicateWithFormat:@"(lastName like[cd] %@) AND (birthday > %@)",
            lastNameSearchString, birthdaySearchDate];
```

この例の`like[cd]`はlike演算子の修正版で、大文字と小文字および発音区別符号を無視します。文字列構文の詳細な説明と使用可能な演算子のリストについては、“プレディケートの書式文字列の構文” （24 ページ）を参照してください。

Important: プレディケートの書式文字列の構文は、正規表現の構文とは異なります。正規表現の構文はICUで定義されています(http://www.icu-project.org/userguide/regexp.htmlを参照してください)。

プレディケート文字列のパーサは空白文字を無視し、キーワードの大文字と小文字を区別しません。また、括弧が入れ子になった式はサポートされます。`printf`形式の書式指定子(`%x`や`%@`など)もサポートされます。“Formatting String Objects”を参照してください。変数は`$`を付加して表します(たとえば、`$VARIABLE_NAME`)。詳細については、“プレディケートテンプレートを使用してプレディケートを作成する” （11 ページ）を参照してください。

パーサは意味的な型チェックを行いません。最良の推測を行って適切な式を作成しようとしますが、特に置換変数がある場合は、実行時エラーが発生する可能性があります。

一般的に、この方法は事前定義されたクエリ語句に最適ですが、変数置換による柔軟性の高さは考慮すべきです。この方法の欠点は、文字列に間違いが含まれないように注意する必要があることです。これらの間違いは実行時まで発見されないでしょう。

文字列定数、変数、およびワイルドカード

式内の文字列定数は引用符で囲む必要があります。一重引用符と二重引用符のどちらも使用できますが、適切な組み合わせで使用する必要があります(つまり、二重引用符(")は一重引用符に(')対応しません)。`%@`を使用して(`firstName like %@`のように)変数置換を使用する場合は、引用符が自動的に追加されます。書式文字列内で文字列定数を使用する場合は、次の例に示すように、引用符を自分で追加する必要があります。

```
NSPredicate *predicate = [NSPredicate
    predicateWithFormat:@"lastName like[c] \"S*\""];
```

ワイルドカードを使用する場合は、自動的に追加される引用符を考慮する必要があります。次の例に示すように、ワイルドカードは置換の前に変数に追加する必要があります。

```
NSString *prefix = @"prefix";
NSString *suffix = @"suffix";
NSPredicate *predicate = [NSPredicate
    predicateWithFormat:@"SELF like[c] %@",
    [[prefix stringByAppendingString:@"*"] stringByAppendingString:suffix]];
BOOL ok = [predicate evaluateWithObject:@"prefixxxxxxsuffix"];
```

この例では、変数置換によりプレディケート文字列が`SELF LIKE[c] "prefix*suffix"`となり、`ok`の値は`YES`になります。一方、次のコードではプレディケート文字列が`SELF LIKE[c] "prefix" * "suffix"`となり、プレディケートの評価で実行時エラーが発生します。

```
predicate = [NSPredicate
    predicateWithFormat:@"SELF like[c] %@*%@", prefix, suffix];
ok = [predicate evaluateWithObject:@"prefixxxxxxsuffix"];
```

また、次のコードは実行時の解析エラーとなります(`Unable to parse the format string "SELF like[c] %@*"`)。

```
predicate = [NSPredicate
```

```
    predicateWithFormat:@"SELF like[c] %@*", prefix];
```

書式文字列内の変数置換と変数式の違いにも注意する必要があります。次のコードでは、右辺が変数式のプレディケートが作成されます。

```
predicate = [NSPredicate
    predicateWithFormat:@"lastName like[c] $LAST_NAME"];
```

変数式の詳細については、“プレディケートテンプレートを使用してプレディケートを作成する”（11 ページ）を参照してください。

ブール値

ブール値を指定する場合、およびブール値が等しいかどうかをテストする場合は、次の例のように指定します。

```
NSPredicate *newPredicate =
    [NSPredicate predicateWithFormat:@"anAttribute == %@", [NSNumber
numberWithBool:aBool]];
NSPredicate *testForTrue =
    [NSPredicate predicateWithFormat:@"anAttribute == YES"];
```

動的なプロパティの名前

文字列変数は、`%@`を使用して書式文字列に置換する際に引用符で囲まれるため、次の例に示すように、`%@`を使用して動的なプロパティの名前を指定することは*できません*。

```
NSString *attributeName = @"firstName";
NSString *attributeValue = @"Adam";
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"%@ like %@",
        attributeName, attributeValue];
```

この場合、プレディケートの書式文字列は`"firstName" like "Adam"`に評価されます。

動的なプロパティの名前を指定する場合は、次のコードに示すように、書式文字列内で`%K`を使用します。

```
predicate = [NSPredicate predicateWithFormat:@"%K like %@",
```

```
        attributeName, attributeValue];
```

この場合、プレディケートの書式文字列は`firstName like "Adam"`に評価されます(`firstName`が引用符で囲まれません)。

プレディケートを直接コード内に作成する

プレディケートと式のインスタンスを直接コード内に作成できます。`NSComparisonPredicate`と`NSCompoundPredicate`には、それぞれ比較プレディケートと複合熟語を簡単に作成できるメソッドが用意されています。`NSComparisonPredicate`には、単純な等価性のテストから独自の機能まで、さまざまな演算子が用意されています。

次の例は、`(revenue >= 1000000) and (revenue < 100000000)`を表すプレディケートを作成する方法を示しています。どちらの比較プレディケートでも、左辺には同じ式を使用しています。

```
NSExpression *lhs = [NSExpression expressionForKeyPath:@"revenue"];


NSExpression *greaterThanRhs = [NSExpression expressionForConstantValue:[NSNumber
 numberWithInt:1000000]];
NSPredicate *greaterThanPredicate = [NSComparisonPredicate
    predicateWithLeftExpression:lhs
    rightExpression:greaterThanRhs
    modifier:NSDirectPredicateModifier
    type:NSGreaterThanOrEqualToPredicateOperatorType
    options:0];


NSExpression *lessThanRhs = [NSExpression expressionForConstantValue:[NSNumber
numberWithInt:100000000]];
NSPredicate *lessThanPredicate = [NSComparisonPredicate
    predicateWithLeftExpression:lhs
    rightExpression:lessThanRhs
    modifier:NSDirectPredicateModifier
    type:NSLessThanPredicateOperatorType
    options:0];


NSCompoundPredicate *predicate = [NSCompoundPredicate andPredicateWithSubpredicates:
```

```
    @[greaterThanPredicate, lessThanPredicate]];
```

一目でわかるように、この方法の欠点は記述しなければならないコードの量が多いことです。利点は、実行時でなければな見つからないようなスペルミスやその他の入力ミスが発生しにくいこと、および文字列の解析に依存するよりも高速であることです。

この方法は、プレディケートの作成自体が動的である場合(プレディケートビルダなどを使用する場合)に最も効果的です。

プレディケートテンプレートを使用してプレディケートを作成する

プレディケートテンプレートは、簡単で使いやすい代わりに間違いの起こりやすい書式文字列に基づく方法と、コードのみを大量に記述して処理する方法の中間に当たります。プレディケートテンプレートは、単純に変数式を含むプレディケートです(Core Dataフレームワークを使用する場合は、Xcodeの設計ツールを使用して、フェッチ要求のプレディケートテンプレートをモデルに追加できます。"Managed Object Models"を参照してください)。次の例は、書式文字列を使用して、左辺が変数式のプレディケートを作成しています。

```
NSPredicate *predicateTemplate = [NSPredicate
    predicateWithFormat:@"lastName like[c] $LAST_NAME"];
```

これは、次の例のように変数式を直接作成する場合と同じです。

```
NSExpression *lhs = [NSExpression expressionForKeyPath:@"lastName"];

NSExpression *rhs = [NSExpression expressionForVariable:@"LAST_NAME"];

NSPredicate *predicateTemplate = [NSComparisonPredicate
    predicateWithLeftExpression:lhs
    rightExpression:rhs
    modifier:NSDirectPredicateModifier
    type:NSLikePredicateOperatorType
    options:NSCaseInsensitivePredicateOption];
```

有効なプレディケートを作成して対象オブジェクトを評価するには、`NSPredicate`の`predicateWithSubstitutionVariables:`メソッドを使用して、置換する変数が含まれている辞書を渡します(辞書には、プレディケートで指定されているすべての変数について、キーと値のペアが含まれている必要があります)。

```
NSPredicate *predicate = [predicateTemplate predicateWithSubstitutionVariables:
    [NSDictionary dictionaryWithObject:@"Turner" forKey:@"LAST_NAME"]];
```

この例で返される新しいプレディケートは、`lastName LIKE[c] "Turner"`です。

置換辞書には、プレディケートに指定されているすべての変数についてキーと値のペアが含まれている必要があるため、null値との一致が必要な場合は、次の例に示すように辞書にnull値を指定する必要があります。

```
NSPredicate *predicate = [NSPredicate
    predicateWithFormat:@"date = $DATE"];
predicate = [predicate predicateWithSubstitutionVariables:
    [NSDictionary dictionaryWithObject:[NSNull null] forKey:@"DATE"]];
```

この例で作成されるプレディケートは、`date == <null>`です。

書式文字列の要約

書式文字列内で値の型を区別することは重要です。変数(または置換変数の文字列)を一重引用符または二重引用符で囲むと、書式文字列内の`%@`、`%K`、または`$variable`がリテラルとして解釈され、置換が行われないことにも注意してください。

`@"attributeName == %@"`
: このプレディケートは、キー`attributeName`の値が、実行時に`predicateWithFormat:`への引数として渡されるオブジェクト`%@`の値と同じであるかどうかを確認します。`%@`は、プレディケート内で記述が有効なオブジェクトのプレースホルダになる場合があることに注意してください(`NSDate`、`NSNumber`、`NSDecimalNumber`、または`NSString`のインスタンスなど)。

`@"%K == %@"`
: このプレディケートは、キー`%K`の値が、オブジェクト`%@`の値と同じであるかどうかを確認します。変数は、実行時に`predicateWithFormat:`の引数として渡されます。

`@"name IN $NAME_LIST"`

キー`name`の値が、実行時に`predicateWithSubstitutionVariables:`を使用して渡される変数`$NAME_LIST`(引用符なし)に存在するかどうかを確認するプレディケートのテンプレートです。

`@"'name' IN $NAME_LIST"`

定数値の`'name'`(文字列が引用符で囲まれていることに注意)が、実行時に`predicateWithSubstitutionVariables:`を使用して渡される変数`$NAME_LIST`に存在するかどうかを確認するプレディケートのテンプレートです。

`@"$name IN $NAME_LIST"`

このプレディケートテンプレートでは、`$NAME`と`$NAME_LIST`の両方で、値が(`predicateWithSubstitutionVariables:`を使用して)置換されることが期待されます。

`@"%K == '%@'"`

このプレディケートは、キー`%K`の値が、文字列リテラルの「`%@`」と等しいかどうかを確認します(`%@`が一重引用符で囲まれていることに注意してください)。キー名の`%K`は、実行時に`predicateWithFormat:`の引数として渡されます。

プレディケートを使用する

ここでは、プレディケートの使用方法の概要と、プレディケートの使用がアプリケーションのデータ構造に与える影響について説明します。

プレディケートを評価する

プレディケートを評価するには、`NSPredicate`の`evaluateWithObject:`メソッドを使用して、プレディケートの評価対象となるオブジェクトを渡します。メソッドはブール値を返します。次の例では、結果は`YES`になります。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"SELF IN %@", @[@"Stig",
 @"Shaffiq", @"Chris"]];
BOOL result = [predicate evaluateWithObject:@"Shaffiq"];
```

プレディケートは任意のクラスのオブジェクトで使用できますが、クラスはプレディケートで使用するキーについてキー値コーディングをサポートしている必要があります。

配列でプレディケートを使用する

`NSArray`と`NSMutableArray`には、配列の内容にフィルタを適用するメソッドが用意されています。`NSArray`の`filteredArrayUsingPredicate:`は、指定したプレディケートに一致するレシーバ内のオブジェクトを含む新しい配列を返します。`NSMutableArray`の`filterUsingPredicate:`は、指定したプレディケートでレシーバのコンテンツを評価し、一致するオブジェクトだけを残します。

```
NSMutableArray *array =
    [NSMutableArray arrayWithObjects:@"Nick", @"Ben", @"Adam", @"Melissa", nil];


NSPredicate *bPredicate = [NSPredicate predicateWithFormat:@"SELF beginswith[c]
'a'"];
NSArray *beginWithB = [array filteredArrayUsingPredicate:bPredicate];
// beginWithB contains { @"Adam" }.
```

```
NSPredicate *sPredicate = [NSPredicate predicateWithFormat:@"SELF contains[c] 'e'"];
[array filterUsingPredicate:sPredicate];
// array now contains { @"Nick", @"Ben", @"Melissa" }
```

Core Dataフレームワークを使用する場合は、配列のメソッドを使用すると、フェッチのように永続データストアに対してラウンドトリップを要求することなく、既存のオブジェクトの配列に対してフィルタを効率的に適用できます。

キーパスでプレディケートを使用する

先に説明したように、キーパスを使用してプレディケートでリレーションシップをたどることができます。次の例では、指定した名前の部署に所属している従業員を検索するプレディケートを作成しています(“パフォーマンスに関する考慮事項” （20 ページ）も参照してください)。

```
NSString *departmentName = ... ;
NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"department.name like %@", departmentName];
```

対多のリレーションシップを使用する場合は、プレディケートの作成方法にいくらか違いがあります。たとえば、「Matthew」という名前の従業員が1人でも所属する部署(Department)をフェッチしたい場合は、次のようにANY演算子を使います。

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

指定した金額を超える給与を受け取っている従業員が1人以上所属する部署を検索する場合は、ANY演算子を次のように使用します。

```
float salary = ... ;
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"ANY employees.salary
> %f", salary];
```

null値を使用する

比較プレディケートでは、null (nil)またはNSNullのnull値以外の値はnullと一致しません(つまり、($value == nil)は$valueがnilの場合にYESを返します)。次の例を考えてみましょう。

```
NSString *firstName = @"Ben";

NSArray *array = @[ @{ @"lastName" : "Turner" }];
                    @{ @"firstName" : @"Ben", @"lastName" : @"Ballard",
                       @"birthday", [NSDate dateWithString:@"1972-03-24 10:45:32
+0600"] } ];

NSPredicate *predicate =
    [NSPredicate predicateWithFormat:@"firstName like %@", firstName];
NSArray *filteredArray = [array filteredArrayUsingPredicate:predicate];

NSLog(@"filteredArray: %@", filteredArray);
// Output:
// filteredArray ({birthday = 1972-03-24 10:45:32 +0600; \\
                      firstName = Ben; lastName = Ballard;})
```

プレディケートはキーfirstNameにBenの値を含む辞書に一致しますが、キーfirstNameに値がない辞書には一致しません。次のコードは、同じ内容を日付と「以上」の比較演算子を使用して示しています。

```
NSDate *referenceDate = [NSDate dateWithTimeIntervalSince1970:0];

predicate = [NSPredicate predicateWithFormat:@"birthday > %@", referenceDate];
filteredArray = [array filteredArrayUsingPredicate:predicate];

NSLog(@"filteredArray: %@", filteredArray);
// Output:
// filteredArray: ({birthday = 1972-03-24 10:45:32 +0600; \\
                       firstName = Ben; lastName = Ballard;})
```

nullをテストする

null値との一致が必要な場合は、次のコードに示すように、他の比較に追加して専用のテストを行う必要があります。

```
predicate = [NSPredicate predicateWithFormat:@"(firstName == %@) || (firstName =
nil)", firstName];

filteredArray = [array filteredArrayUsingPredicate:predicate];

NSLog(@"filteredArray: %@", filteredArray);


// Output:

// filteredArray: ( { lastName = Turner; }, { birthday = 1972-03-23 20:45:32 -0800;
 firstName = Ben; lastName = Ballard; }
```

暗黙的に、null値に一致するnullのテストは真を返します。次のコードでは、どちらのプレディケートの評価でもokはYESに設定されます。

```
predicate = [NSPredicate predicateWithFormat:@"firstName = nil"];

BOOL ok = [predicate evaluateWithObject:[NSDictionary dictionary]];


ok = [predicate evaluateWithObject:

    [NSDictionary dictionaryWithObject:[NSNull null] forKey:@"firstName"]];
```

Core Dataでプレディケートを使用する

Core Dataフレームワークを使用している場合は、Core Dataを使用しない場合と同じ方法でプレディケートを使用できます(たとえば、配列にフィルタを適用したり、配列コントローラで使用したりできます)。また、プレディケートをフェッチ要求で制約として使用したり、フェッチ要求のテンプレートを管理オブジェクトモデルに保存したりもできます(“Managed Object Models”を参照してください)。

制限事項: すべてのSQLクエリをプレディケートまたはフェッチ要求に変換できるとは限りません。たとえば、次のようなSQL文を考えます。

```
SELECT t1.name, V1, V2
    FROM table1 t1 JOIN (SELECT t2.name AS V1, count(*) AS V2
        FROM table2 t2 GROUP BY t2.name as V) on t1.name = V.V1
```

このSQL文をフェッチ要求に変換する方法はありません。対象のオブジェクトをフェッチした後、結果を使用して直接計算するか、配列演算子を使用する必要があります。

フェッチ要求

ターゲットエンティティのプロパティに一致するプレディケートを作成し(キーパスを使用してリレーションシップをたどることができます)、プレディケートをフェッチ要求に関連付けます。要求が実行されるときに、プレディケートで指定した条件に一致するオブジェクトがあれば、これらのオブジェクトを含む配列が返されます。次の例では、プレディケートを使用して、給与が指定した金額を超えている従業員を検索しています。

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

NSNumber *salaryLimit = <#A number representing the limit#>;
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"salary > %@",
salaryLimit];
[request setPredicate:predicate];
NSError *error;
NSArray *array = [managedObjectContext executeFetchRequest:request error:&error];
```

オブジェクトコントローラ

Cocoaバインディングを使用している場合は、オブジェクトコントローラ(`NSObjectController`や`NSArrayController`のインスタンスなど)のフェッチプレディケートを指定できます。XcodeのAttributes Inspectorでプレディケートエディタのテキストフィールドにプレディケートを直接入力するか、`setFetchPredicate:`を使用してプログラムで設定します。プレディケートは、コントローラがフェッチを実行したときに返される結果を制限するために使用されます。`NSObjectController`オブジェク

トを使用している場合は、コントローラのコンテンツにするオブジェクトを一意に識別するフェッチを指定します。たとえば、コントローラのエンティティがDepartment(部署)である場合は、`name like "Engineering"`などのプレディケートを使用します。

正規表現を使用する

`MATCHES`演算子は、次の例に示すようにICUの正規表現パッケージを使用します。

```
NSArray *array = @[@"TATACCATGGGCCATCATCATCATCATCATCATCATCATCACAG",
                   @"CGGGATCCCTATCAAGGCACCTCTTCG", @"CATGCCATGGATACCAACGAGTCCGAAC",
                     @"CAT", @"CATCATCATGTCT", @"DOG"];

// find strings that contain a repetition of at least 3 'CAT' sequences,
// but not followed by a further 'CA'
NSPredicate *catPredicate =
    [NSPredicate predicateWithFormat:@"SELF MATCHES '.*(CAT){3,}(?!CA).*'"];

NSArray *filteredArray = [array filteredArrayUsingPredicate:catPredicate];
// filteredArray contains just 'CATCATCATGTCT'
```

ICUの仕様に従い、パターンのセット内では正規表現のメタ文字は有効ではありません。たとえば、正規表現の`\d{9}[\dxX]`は、有効なISBN番号(10桁の数字、または9桁の数字とXからなる文字列)に一致*しません*。これはパターンのセット(`[\dxX]`)にメタ文字(`\d`)が含まれているためです。この場合は、次のコード例に示すように`OR`式を記述します。

```
NSArray *isbnTestArray = @[@"123456789X", @"987654321x", @"1234567890", @"12345X",
 @"1234567890X"];
NSPredicate *isbnPredicate =
    [NSPredicate predicateWithFormat:@"SELF MATCHES '\\\\d{10}|\\\\d{9}[Xx]'"];

NSArray *isbnArray = [isbnTestArray filteredArrayUsingPredicate:isbnPredicate];
// isbnArray contains (123456789X, 987654321x, 1234567890)
```

パフォーマンスに関する考慮事項

複合プレディケートは、処理量が最小となるように作成してください。特に、正規表現の照合は負荷の高い演算です。したがって複合プレディケートでは、簡単なテストを正規表現より前に実行します。たとえば、次のような例を考えます。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"( title matches .*mar[1-10] ) OR ( type = 1 )"];
```

この場合は、次のように記述してください。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"( type = 1 ) OR ( title matches .*mar[1-10] )"];
```

2つ目の例では、最初の句が偽の場合にのみ正規表現が評価されます。

ジョインを使用する

一般的に、ジョイン(複数のリレーションシップにまたがるクエリ)も負荷の高い演算です。可能であれば、使用を避けてください。対1のリレーションシップをテストするときに、リレーションシップのソースオブジェクト(またはそのオブジェクトID)がすでにわかっている場合や、ソースオブジェクトを簡単に取得できる場合は、オブジェクトのプロパティをテストするよりも、その等価性をテストする方が効率的です。たとえば、次のような記述を考えます。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"department.name like %@", [department name]];
```

この場合は、次のように記述する方が効率的です。

```
NSPredicate *predicate = [NSPredicate predicateWithFormat:
        @"department == %@", department];
```

プレディケートに複数の式が含まれる場合も、一般的にはジョインを避けるように作成する方が効率的です。たとえば、`@"firstName beginswith[cd] 'Matt' AND (ANY directreports.paygrade <= 7)"`は、`@"(ANY directreports.paygrade <= 7) AND (firstName beginswith[cd] 'Matt')"`よりも効率的です。前者では最初のテストに成功しなければジョインが実行されません。

データの構造を決定する

状況によっては、データの表現方法とプレディケートの使用が相互に影響する場合があります。アプリケーションでプレディケートを使用する予定の場合は、一般的なクエリ操作のパターンが、構築するデータの構造に影響を与える可能性があります。Core Dataでは、エンティティとエンティティクラスのマッピングを指定しますが、永続ストアに基礎構造を作成するレベルは不透明です。それでも、エンティティとそのプロパティは管理できます。

また、ジョインは負荷が高くなりやすいだけでなく、柔軟性にも制限を与える場合があります。したがって、データを非正規化する方が適切な場合もあります。一般的に、クエリが頻繁に発行されるケースでは、オブジェクトのサイズを大きくして、正しいオブジェクトを簡単に検索できるようにする(したがって、メモリに読み込む量を減らす)ことで、効率がよくなる場合もあります。

Cocoaバインディングでプレディケートを使用する

OS Xでは、配列コントローラにプレディケートを設定して、コンテンツの配列にフィルタを適用できます。プレディケートをコード内に(`setFilterPredicate:`を使用して)設定できます。また、配列コントローラの`filterPredicate`バインディングを、`NSPredicate`オブジェクトを返すメソッドにバインドすることもできます。メソッドを実装するオブジェクトは、ファイルの所有者か別のコントローラオブジェクトになります。プレディケートを変更する場合は、配列コントローラが自身を更新するように、キー値監視に準拠した方法で行う必要があります(『*Key-Value Observing Programming Guide*』を参照してください)。

また、`NSSearchField`オブジェクトの`predicate`バインディングを配列コントローラの`filterPredicate`にバインドすることもできます。検索フィールドの`predicate`バインディングは、"Binding Types"で説明しているように、複数値のバインディングです。

NSPredicateとSpotlightのクエリ文字列の比較

SpotlightとNSPredicateクラスは、どちらもクエリ文字列の構文を実装します。両者は似ていますが、いくつかの点で違いがあります。共通の機能のサブセットを使用していれば、一方のクエリ文字列を他方の文字列形式に変換することが可能です。

SpotlightとNSPredicate

NSMetadataQueryクラス(Spotlightに対するCocoaのインターフェイス)は、APIでNSPredicateを使用します。この点を除けば、SpotlightとNSPredicateの間に関係はありません。Spotlightのクエリ文字列構文はNSPredicateのクエリ文字列構文と似ていますが、いくつか違いがあります。両方のAPIで認識される構文を使用すれば、クエリ文字列を相互に変換することができます。Spotlightのクエリ構文は、NSPredicateのクエリ構文のサブセットです。Spotlightのクエリ式の構文については、"File Metadata Query Expression Syntax"を参照してください。NSPredicateの文字列構文の詳細については、"プレディケートの書式文字列の構文" （24 ページ）を参照してください。

Spotlightの比較句は、"KEY operator VALUE"という形式にする必要があります。"VALUE operator KEY"の形式は使用できません。また、SpotlightでVALUEに指定できる属性の種類は、NSPredicateで使用できる属性よりも限られています。この制限の結果として、Spotlightのクエリではリテラル文字列を必ずしも引用符で囲む必要はありません。VALUEが文字列である場合、特殊な演算子を適用する必要がなければ、引用符を省略できます。NSPredicateのクエリ文字列では、結果があいまいになるため引用符は省略できません。

Spotlightのクエリで大文字と小文字および発音区別符号を示す構文は、NSPredicateの構文と異なります。Spotlightでは、比較文字列の後にマーカーを付加します(たとえば、"myAttribute == 'foo'cd")。NSPredicateの文字列では、like演算子を使用し、マーカーを"[ ]"で囲んで比較文字列の前(演算子の後)に置きます(たとえば、"myAttribute like[cd] 'foo'")。どちらの場合も、'cd'は大文字と小文字および発音区別符号を区別しないことを表します。Spotlightでは限定子を値に付加し、NSPredicateでは値を演算子に付加します。

MDQuery演算子を、NSPredicateオブジェクトの"KEY operator VALUE"文字列でVALUEとして使用することはできません。たとえば、「～のサブ文字列である」という式を作成する場合、Spotlightでは"myAttribute = '*foo*'"のように指定します。NSPredicateの文字列では、contains演算子を使用して"myAttribute contains 'foo'"のように指定します。SpotlightではGLOB句のような式を使用し、NSPredicateで使用する演算子とは異なります。

比較式で左辺のキーとして「*」を使用すると、Spotlightでは「項目内の任意のキー」という意味になり、==のみで使用できます。. この式をNSPredicateオブジェクトで使用する場合は、NSMetadataQueryオブジェクトと組み合わせる必要があります。

FinderのSpotlight検索からプレディケートの書式文字列を作成する

Finderの検索からプレディケートの書式文字列を作成できます。検索を実行して保存した後、保存したフォルダを選択して「Show Info」を選択します。「Info」パネルに、Spotlightで使用されたクエリが表示されます。ただし、NSPredicateの書式文字列と、Finderに保存されている書式文字列には、いくつか違いがあることに注意してください。Finderの文字列は次のようになります。

```
(((* = "FooBar*"wcd) || (kMDItemTextContent = "FooBar*"cd))
    && (kMDItemContentType != com.apple.mail.emlx)
    && (kMDItemContentType != public.vcard))
```

一般的に、Spotlightのクエリをプレディケートの書式文字列に変換する際に必要な作業は、プレディケートの先頭が*でないかどうかを確認することだけです(これは、プレディケートを解析する際にNSMetadataQueryでサポートされません)。ワイルドカードを使用する必要がある場合は、次の例に示すようにLIKEを使用します。

```
((kMDItemTextContent LIKE[cd] "FooBar")
    && (kMDItemContentType != "com.apple.mail.emlx")
    && (kMDItemContentType != "public.vcard"))
```

プレディケートの書式文字列の構文

ここではプレディケート文字列の構文と、プレディケートパーサの特徴について説明します。

パーサ文字列と正規表現エンジンに渡される文字列式は異なります。ここでは正規表現エンジンの構文ではなく、パーサのテキストについて説明します。

パーサの基本事項

プレディケート文字列のパーサは空白文字を無視し、キーワードの大文字と小文字を区別しません。また、括弧が入れ子になった式はサポートされます。パーサは意味的な型チェックを行いません。

変数はドル記号(`$`)を付加して表します(たとえば、`$VARIABLE_NAME`)。疑問符(`?`)は、有効なパーサのトークンではありません。

書式文字列は、`%x`などの`printf`形式の書式指定子をサポートします(“Formatting String Objects”を参照してください)。重要な書式指定子として、`%@`と`%K`の2つがあります。

- `%@`は、暗黙的な型指定によるオブジェクト値(通常は文字列、数値、または日付)への引数置換を表します。
- `%K`は、暗黙的な型指定によるキーパスへの引数置換を表します。

文字列変数は、`%@`書式指定子を使用して文字列に置換される際に引用符で囲まれます。動的なプロパティの名前を指定する場合は、次の例に示すように、書式文字列内で`%K`を使用します。

```
NSString *attributeName  = @"firstName";
NSString *attributeValue = @"Adam";
NSPredicate *predicate   = [NSPredicate predicateWithFormat:@"%K like %@",
        attributeName, attributeValue];
```

この場合、プレディケートの書式文字列は`firstName like "Adam"`に評価されます。

変数(または置換変数の文字列)を一重引用符または二重引用符で囲むと、書式文字列内の`%@`、`%K`、または`$variable`がリテラルとして解釈され、置換は行われません。次の例では、プレディケートの書式文字列は`firstName like "%@"`に評価されます(`%@`が一重引用符で囲まれます)。

```
NSString *attributeName = @"firstName";
NSString *attributeValue = @"Adam";
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"%K like '%@'",
        attributeName, attributeValue];
```

Important: %@書式指定子は、式を表現する場合にのみ使用してください。プレディケート全体を表現する目的では使用しないでください。

書式指定子を使用してプレディケート全体を表現しようとすると、例外が発生します。

基本的な比較

=, ==
: 左辺の式と右辺の式が等しい。

>=, =>
: 左辺の式が右辺の式より大きいか等しい。

<=, =<
: 左辺の式が右辺の式より小さいか等しい。

>
: 左辺の式が右辺の式より大きい。

<
: 左辺の式が右辺の式より小さい。

!=, <>
: 左辺の式と右辺の式が等しくない。

BETWEEN
: 左辺の式が、右辺で指定した値の範囲内にあるか、上限または下限の値に等しい。

  右辺は、上限と下限を表す2つの値の配列です(配列は正しい順番で指定する必要があります)。たとえば、`1 BETWEEN { 0 , 33 }`や`$INPUT BETWEEN { $LOWER, $UPPER }`のように指定します。

  Objective-Cでは、次の例に示すようにBETWEENプレディケートを作成できます。

```
NSPredicate *betweenPredicate =
```

```
    [NSPredicate predicateWithFormat: @"attributeName BETWEEN %@", @[@1, @10]];
```

この場合、次の例に示すように`( ( 1 <= attributeValue ) && ( attributeValue <= 10 ) )`に一致するプレディケートが作成されます。

```
NSPredicate *betweenPredicate =
    [NSPredicate predicateWithFormat: @"attributeName BETWEEN %@", @[@1, @10]];

NSDictionary *dictionary = @{ @"attributeName" : @5 };

BOOL between = [betweenPredicate evaluateWithObject:dictionary];
if (between) {
    NSLog(@"between");
}
```

ブール値のプレディケート

`TRUEPREDICATE`
: 常に`TRUE`に評価されるプレディケート。

`FALSEPREDICATE`
: 常に`FALSE`に評価されるプレディケート。

基本的な複合プレディケート

`AND, &&`
: 論理AND。

`OR, ||`
: 論理OR。

`NOT, !`
: 論理NOT。

文字列の比較

文字列の比較では、デフォルトで大文字と小文字および発音区別符号が区別されます。キー文字の`c`と`d`を角括弧内に指定して演算子を修飾すると、それぞれ大文字と小文字および発音区別符号を区別しないように指定することができます。たとえば、`firstName BEGINSWITH[cd] $FIRST_NAME`のように指定します。

`BEGINSWITH`
: 左辺の式の先頭が右辺の式に等しい。

`CONTAINS`
: 左辺の式に右辺の式が含まれている。

`ENDSWITH`
: 左辺の式の末尾が右辺の式に等しい。

`LIKE`
: 左辺の式と右辺の式が等しい。`?`と`*`をワイルドカードとして使用できます。`?`は`1`文字と一致し、`*`は`0`文字以上と一致します。

`MATCHES`
: ICU v3に従って`regex`形式の比較を使用したときに、左辺の式と右辺の式が等しい(詳細については、『ICU User Guide』の「Regular Expressions」を参照してください)。

`UTI-CONFORMS-TO`
: この演算子の左辺の引数は、照合する統一型識別子(UTI)に評価される式です。右辺の引数はUTIに評価される式です。左辺の式から返されたUTIが右辺の式から返されたUTIに準拠している場合は、比較が`TRUE`に評価されます。指定した型にどの型が準拠するかについては、『*Uniform Type Identifiers Reference*』の"System-Declared Uniform Type Identifiers"を参照してください。カスタムUTIへの準拠を宣言する方法については、『*Uniform Type Identifiers Overview*』の"Declaring New Uniform Type Identifiers"を参照してください。

  `A UTI-CONFORMS-TO B`という句では、`UTTypeConformsTo`メソッドを次のように使用する場合と同じ結果が得られます。

```
UTTypeConformsTo (A, B)
```

  アプリケーション拡張機能の項目の付加情報(`NSExtensionItem`型)を評価する場合は、次のような文を使用します。

```
SUBQUERY (
    extensionItems,
    $extensionItem,
    SUBQUERY (
```

```
        $extensionItem.attachments,
        $attachment,
        ANY $attachment.registeredTypeIdentifiers UTI-CONFORMS-TO "com.adobe.pdf"
    ).@count == $extensionItem.attachments.@count
).@count == 1
```

`UTI-EQUALS`
: この演算子の左辺の引数は、照合する統一型識別子(UTI)に評価される式です。右辺の引数はUTIに評価される式です。左辺の式から返されたUTIが右辺の式から返されたUTIと等しい場合は、比較が`TRUE`に評価されます。

  `A UTI-EQUALS B`という句では、`UTTypeEqual`メソッドを次のように使用する場合と同じ結果が得られます。

```
UTTypeEqual (A, B)
```

  `UTI-CONFORMS-TO`のコード例を参照してください。演算子を`UTI-EQUALS`に置き換えるだけで、同様に動作します。

集約演算 (Aggregate Operations)

`ANY`, `SOME`
: 後に続く式の任意の要素を指定します。たとえば `ANY children.age < 18`.

`ALL`
: 後に続く式のすべての要素を指定します。たとえば `ALL children.age < 18`.

`NONE`
: 後に続く式のいずれの要素も指定しません。たとえば`NONE children.age < 18`と記述します。`NOT (ANY ...)`と論理的に等価です。

`IN`
: SQLのIN演算子と等価で、左辺の項が右辺に指定したコレクション内に存在している必要があります。

  たとえば`name IN { 'Ben', 'Melissa', 'Nick' }`と記述します。コレクションには配列、集合、または辞書を指定できます。辞書を指定した場合は、辞書の値が使用されます。

  Objective-Cでは、次の例に示すようにINプレディケートを作成できます。

```
NSPredicate *inPredicate =
            [NSPredicate predicateWithFormat: @"attribute IN %@", aCollection];
```

この場合、aCollectionはNSArray、NSSet、またはNSDictionaryのインスタンス、もしくは対応する可変クラスのインスタンスです。

`array[index]`
配列arrayの、指定したインデックス位置の要素を指定します。

`array[FIRST]`
配列arrayの先頭の要素を指定します。

`array[LAST]`
配列arrayの末尾の要素を指定します。

`array[SIZE]`
配列arrayのサイズを指定します。

識別子 (Identifiers)

C形式の識別子
予約語以外のC形式の識別子。

#記号
ユーザ識別子内で予約語をエスケープするために使用します。

[\]{8進数}{3}
8進数(\の後に続く3桁の数字)をエスケープするために使用します。

[\][xX]{16進数}{2}
16進数(\xまたは\Xの後に続く2桁の数字)をエスケープするために使用します。

[\][uU]{16進数}{4}
ユニコード数(\uまたは\Uの後に続く4桁の数字)をエスケープするために使用します。

リテラル

一重引用符と二重引用符のどちらでも同じ結果になりますが、種類の異なる引用符を対にすることはできません。たとえば、"abc"と'abc'は同じですが、"a'b'c"はa、'b'、およびcを空白区切りでジョインした場合と同じになります。

`FALSE`, `NO`
: 論理値の偽。

`TRUE`, `YES`
: 論理値の真。

`NULL`, `NIL`
: null値。

`SELF`
: 評価されるオブジェクトを表します。

`"text"`
: 文字列。

`'text'`
: 文字列。

カンマ区切りのリテラル配列
: たとえば`{ 'comma', 'separated', 'literal', 'array' }`と記述します。

標準の整数および固定小数点の記法
: たとえば、`1`、`27`、`2.71828`、`19.75`などです。

累乗を使用する浮動小数点記法
: たとえば、`9.2e-5`です。

`0x`
: 16進数を表すプリフィックス。

`0o`
: 8進数を表すプリフィックス。

`0b`
: 2進数を表すプリフィックス。

予約語

次の語句は予約されています。

`AND`, `OR`, `IN`, `NOT`, `ALL`, `ANY`, `SOME`, `NONE`, `LIKE`, `CASEINSENSITIVE`, `CI`, `MATCHES`, `CONTAINS`, `BEGINSWITH`, `ENDSWITH`, `BETWEEN`, `NULL`, `NIL`, `SELF`, `TRUE`, `YES`, `FALSE`, `NO`, `FIRST`, `LAST`, `SIZE`, `ANYKEY`, `SUBQUERY`, `CAST`, `TRUEPREDICATE`, `FALSEPREDICATE`, `UTI-CONFORMS-TO`, `UTI-EQUALS`

CocoaのプレディケートのBNF定義

ここでは、Cocoaのプレディケートをバッカスナウア記法で定義します。

NSPredicate

```
NSPredicate ::= NSComparisonPredicate | NSCompoundPredicate
    | "(" NSPredicate ")" | TRUEPREDICATE | FALSEPREDICATE
```

NSCompoundPredicate

```
NSCompoundPredicate ::= NSPredicate "AND" NSPredicate
    | NSPredicate "OR" NSPredicate
    | "NOT" NSPredicate
```

NSComparisonPredicate

```
NSComparisonPredicate ::= expression operation expression
    | aggregate_qualifier NSComparisonPredicate
```

演算 (Operations)

`CONTAINS`と`IN`は、どちらも引数の型に応じて集約演算子および文字列演算子として動作します。

```
operation ::= "=" | "!=" | "<" | ">" | "<=" | ">="
    | BETWEEN
    | aggregate_operations [ "[" string_options "]" ]
```

```
aggregate_operations ::= CONTAINS | IN | string_operations

string_operations ::= BEGINSWITH | ENDSWITH | LIKE | MATCHES

string_options ::= c | d | cd
```

集約限定子 (Aggregate Qualifier)

```
aggregate_qualifier ::= ANY | ALL | NONE | SOME
```

式 (Expression)

```
expression ::= "(" expression ")"
    | binary_expression
    | function_expression
    | assignment_expression
    | index_expression
    | keypath_expression
    | value_expression
            10.6: ternary_expression, function, union
```

値式 (Value Expression)

```
value_expression ::= literal_value | literal_aggregate
```

リテラル値 (Literal Value)

```
literal_value ::= string_value
```

```
    | numeric_value
    | predicate_argument
    | predicate_variable
    | NULL
    | TRUE
    | FALSE
    | SELF
```

文字列値 (String Value)

```
string_value ::= "text" | 'text'
```

プレディケートの引数 (Predicate Argument)

```
predicate_argument ::= "%" format_argument
```

書式の引数 (Format Argument)

```
format_argument ::= "@" | "%" | "K"
    printf style conversion character
```

プレディケートの変数 (Predicate Variable)

```
predicate_variable ::= "$" identifier
```

キーパス式 (Keypath Expression)

```
keypath_expression ::= identifier | "@" identifier
```

```
    | expression "." expression
```

リテラルの集約 (Literal Aggregate)

```
literal_aggregate ::= "{" [ expression [ "," expression ... ] ] "}"
```

インデックス式 (Index Expression)

```
index_expression ::= array_expression "[" integer_expression "]"
    | dictionary_expression   "[" expression "]"
    | aggregate_expression "[" FIRST "]"
    | aggregate_expression "[" LAST "]"
    | aggregate_expression "[" SIZE "]"
```

集約式 (Aggregate Expression)

```
aggregate_expression ::= array_expression | dictionary_expression
```

代入式 (Assignment Expression)

```
assignment_expression ::= predicate_variable ":=" expression
```

バイナリ式 (Binary Expression)

```
binary_expression ::= expression binary_operator expression
    | "-" expression
```

バイナリ演算子 (Binary Operator)

```
binary_operator ::= "+" | "-" | "*" | "/" | "**"
```

関数式 (Function Expression)

```
function_expression ::= function_name "(" [ expression [ "," expression ... ] ]
")"
```

関数名 (Function Name)

```
function_name ::= "sum" | "count" | "min" | "max"
    | "average" | "median" | "mode" | "stddev"
    | "sqrt" | "log" | "ln" | "exp"
    | "floor" | "ceiling" | "abs" | "trunc"
    | "random" | "randomn" | "now"
```

配列式 (Array Expression)

```
array_expression ::= any expression that evaluates to an NSArray object
```

辞書式 (Dictionary Expression)

```
dictionary_expression ::= any expression that evaluates to an NSDictionary object
```

整数式 (Integer Expression)

```
integer_expression ::= any expression that evaluates to an integral value
```

数値 (Numeric Value)

```
numeric_value ::= C style numeric constant
```

識別子 (Identifier)

```
identifier ::= C style identifier | "#" reserved_word
```

書類の改訂履歴

この表は「*プレディケートプログラミングガイド*」の改訂履歴です。

日付	メモ
2014-09-17	新しいUTI-CONFORMS-TOおよびUTI-EQUALSプレディケート演算子の説明を追加しました。 一部の説明をわかりやすく修正しました。
2010-06-14	「はじめに」の内容をわかりやすく改訂しました。「FinderのSpotlight検索からプレディケートの書式文字列を作成する」の細かい間違いを訂正しました。
2009-11-17	Core Data SQLiteストアでサポートされる演算の説明を訂正しました。
2009-02-03	iOS 3.0用に更新しました。
2008-10-15	BETWEEN演算子の例をわかりやすく修正しました。
2006-10-03	「プレディケートの書式文字列の構文」の変数置換に関する問題をわかりやすく修正し、ブール値の例を追加しました。
2006-05-23	LIKE演算子でのワイルドカードの使用方法をわかりやすく修正しました。
2006-04-04	Core Dataモデルベースのフェッチ要求に関する説明を、『Core Data Programming Guide』に移動しました。

日付	メモ
2006-03-08	プレディケートで使用するキーについて、キー値コーディングへの準拠の必要性をわかりやすく修正しました。
2006-01-10	誤字を訂正しました。
2005-11-09	細かな誤字を訂正しました。
2005-10-04	リレーションシップの例をわかりやすく修正しました。正規表現に関するセクションを追加しました。
2005-08-11	NSPredicateとSpotlightのクエリについて比較情報を追加しました。Cocoaバインディングおよび Core Dataでのプレディケートの使用方法についての説明を追加しました。
2005-07-07	細かな誤字を訂正しました。
2005-04-29	新規資料。Cocoaでクエリを指定する方法を解説。

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

Apple Japan
〒106-6140 東京都港区六本木
6丁目10番1号 六本木ヒルズ
http://www.apple.com/jp

Offline copy. Trademarks go here.