NSPredicate

Written by Mattt — July 15th, 2013

`NSPredicate` is a Foundation class that specifies how data should be fetched or filtered. Its query language, which is like a cross between a SQL `WHERE` clause and a regular expression, provides an expressive, natural language interface to define logical conditions on which a collection is searched.

It's easier to show `NSPredicate` in use, rather than talk about it in the abstract, so we're going to revisit the example data set used in the `NSSortDescriptor` articlehttps://nshipster.com/nssortdescriptor/:

firstName	lastName	age
Alice	Smith	24
Bob	Jones	27
Charlie	Smith	33
Quentin	Alberts	31

```swift
@objcMembers class Person: NSObject {
    let firstName: String
    let lastName: String
    let age: Int

    init(firstName: String, lastName: String, age: Int) {
        self.firstName = firstName
        self.lastName = lastName
        self.age = age
    }

    override var description: String {
        return "\(firstName) \(lastName)"
    }
}

let alice = Person(firstName: "Alice", lastName: "Smith", age: 24)
let bob = Person(firstName: "Bob", lastName: "Jones", age: 27)
let charlie = Person(firstName: "Charlie", lastName: "Smith", age: 33)
let quentin = Person(firstName: "Quentin", lastName: "Alberts", age: 31)
let people = [alice, bob, charlie, quentin] as NSArray

let bobPredicate = NSPredicate(format: "firstName = 'Bob'")
let smithPredicate = NSPredicate(format: "lastName = %@", "Smith")
let thirtiesPredicate = NSPredicate(format: "age >= 30")

people.filtered(using: bobPredicate)
// ["Bob Jones"]

people.filtered(using: smithPredicate)
// ["Alice Smith", "Charlie Smith"]

people.filtered(using: thirtiesPredicate)
// ["Charlie Smith", "Quentin Alberts"]
```

Using `NSPredicate` with Collections

Foundation provides methods to filter `NSArray` / `NSMutableArray` & `NSSet` / `NSMutableSet` with predicates.

Immutable collections, `NSArray` & `NSSet`, have the methods `filteredArrayUsingPredicate:` and `filteredSetUsingPredicate:` which return an immutable collection by evaluating a predicate on the receiver.

Mutable collections, `NSMutableArray` & `NSMutableSet` have the method `filterUsingPredicate:`, which removes any objects that evaluate to `FALSE` when running the predicate on the receiver.

`NSDictionary` can use predicates by filtering its keys or values (both `NSArray` objects). `NSOrderedSet` can either create new ordered sets from a filtered `NSArray` or `NSSet`, or alternatively, `NSMutableSet` can simply `removeObjectsInArray:`, passing objects filtered with the *negated* predicate.

Using `NSPredicate` with Core Data

`NSFetchRequest` has a `predicate` property, which specifies the logical conditions under which managed objects should be retrieved. The same rules apply, except that predicates are evaluated by the persistent store coordinator within a managed object context, rather than collections being filtered in-memory.

Predicate Syntax

Substitutions

> - `%@` is a var arg substitution for an object value—often a string, number, or date.
> - `%K` is a var arg substitution for a key path.

Swift Objective-C

```
let ageIs33Predicate = NSPredicate(format: "%K = %@", "age", "33")

people.filtered(using: ageIs33Predicate)
// ["Charlie Smith"]
```

> - `$VARIABLE_NAME` is a value that can be substituted with `NSPredicate -predicateWithSubstitutionVariables:`.

Swift Objective-C

```
let namesBeginningWithLetterPredicate = NSPredicate(format: "(firstName BEGIN

people.filtered(using: namesBeginningWithLetterPredicate.withSubstitutionVari
// ["Alice Smith", "Quentin Alberts"]
```

Basic Comparisons

- `=`, `==`: The left-hand expression is equal to the right-hand expression.
- `>=`, `=>`: The left-hand expression is greater than or equal to the right-hand expression.
- `<=`, `=<`: The left-hand expression is less than or equal to the right-hand expression.
- `>`: The left-hand expression is greater than the right-hand expression.
- `<`: The left-hand expression is less than the right-hand expression.
- `!=`, `<>`: The left-hand expression is not equal to the right-hand expression.
- `BETWEEN`: The left-hand expression is between, or equal to either of, the values specified in the right-hand side. The right-hand side is a two value array (an array is required to specify order) giving upper and lower bounds. For example, `1 BETWEEN { 0 , 33 }`, or `$INPUT BETWEEN { $LOWER, $UPPER }`.

Basic Compound Predicates

- `AND`, `&&`: Logical `AND`.
- `OR`, `||`: Logical `OR`.
- `NOT`, `!`: Logical `NOT`.

String Comparisons

String comparisons are by default case and diacritic sensitive. You can modify an operator using the key characters c and d within square braces to specify case and diacritic insensitivity respectively, for example firstName BEGINSWITH[cd] $FIRST_NAME.

- `BEGINSWITH` : The left-hand expression begins with the right-hand expression.
- `CONTAINS` : The left-hand expression contains the right-hand expression.
- `ENDSWITH` : The left-hand expression ends with the right-hand expression.
- `LIKE` : The left hand expression equals the right-hand expression: `?` and `*` are allowed as wildcard characters, where `?` matches 1 character and `*` matches 0 or more characters.
- `MATCHES` : The left hand expression equals the right hand expression using a regex-style comparison according to ICU v3 (for more details see the ICU User Guide for Regular Expressions).

Aggregate Operations

Relational Operations

- `ANY` , `SOME` : Specifies any of the elements in the following expression. For example, `ANY children.age < 18` .
- `ALL` : Specifies all of the elements in the following expression. For example, `ALL children.age < 18` .
- `NONE` : Specifies none of the elements in the following expression. For example, `NONE children.age < 18` . This is logically equivalent to `NOT (ANY ...)` .
- `IN` : Equivalent to an SQL `IN` operation, the left-hand side must appear in the collection specified by the right-hand side. For example, `name IN { 'Ben', 'Melissa', 'Nick' }` .

Array Operations

> - `array[index]` : Specifies the element at the specified index in `array`.
> - `array[FIRST]` : Specifies the first element in `array`.
> - `array[LAST]` : Specifies the last element in `array`.
> - `array[SIZE]` : Specifies the size of `array`.

Boolean Value Predicates

> - `TRUEPREDICATE` : A predicate that always evaluates to `TRUE`.
> - `FALSEPREDICATE` : A predicate that always evaluates to `FALSE`.

NSCompoundPredicate

We saw that `AND` & `OR` can be used in predicate format strings to create compound predicates. However, the same can be accomplished using an `NSCompoundPredicate`.

For example, the following predicates are equivalent:

Swift Objective-C

```
NSCompoundPredicate(
    type: .and,
    subpredicates: [
        NSPredicate(format: "age > 25"),
        NSPredicate(format: "firstName = %@", "Quentin")
    ]
)

NSPredicate(format: "(age > 25) AND (firstName = %@)", "Quentin")
```

While the syntax string literal is certainly easier to type, there are occasions where you may need to combine existing predicates. In these cases, `NSCompoundPredicate -andPredicateWithSubpredicates:` & `-orPredicateWithSubpredicates:` is the way to go.

NSComparisonPredicate

Similarly, if after reading last week's articlehttps://nshipster.com/nsexpression/ you now find yourself with more `NSExpression` objects than you know what to do with, `NSComparisonPredicate` can help you out.

Like `NSCompoundPredicate`, `NSComparisonPredicate` constructs an `NSPredicate` from subcomponents—in this case, `NSExpression`s on the left and right hand sides. Analyzing its class constructor provides a glimpse into the way `NSPredicate` format strings are parsed:

Objective-C Swift

```
+ (NSPredicate *)predicateWithLeftExpression:(NSExpression *)lhs
                             rightExpression:(NSExpression *)rhs
                                    modifier:(NSComparisonPredicateModifier)m
                                        type:(NSPredicateOperatorType)type
                                     options:(NSUInteger)options
```

Parameters

- `lhs`: The left hand expression.
- `rhs`: The right hand expression.
- `modifier`: The modifier to apply. (`ANY` or `ALL`)
- `type`: The predicate operator type.
- `options`: The options to apply. For no options, pass `0`.

NSComparisonPredicate Types

```swift
enum NSComparisonPredicate.Operator: UInt {
    case lessThan
    case lessThanOrEqualTo
    case greaterThan
    case greaterThanOrEqualTo
    case equalTo
    case notEqualTo
    case matches
    case like
    case beginsWith
    case endsWith
    case `in`
    case customSelector
    case contains
    case between
}
```

`NSComparisonPredicate` Options

- `NSCaseInsensitivePredicateOption`: A case-insensitive predicate. You represent this option in a predicate format string using a [c] following a string operation (for example, `"NeXT" like[c] "next"`).
- `NSDiacriticInsensitivePredicateOption`: A diacritic-insensitive predicate. You represent this option in a predicate format string using a [d] following a string operation (for example, `"naïve" like[d] "naive"`).
- `NSNormalizedPredicateOption`: Indicates that the strings to be compared have been preprocessed. This option supersedes NSCaseInsensitivePredicateOption and NSDiacriticInsensitivePredicateOption, and is intended as a performance optimization option. You represent this option in a predicate format string using a [n] following a string operation (for example, `"WXYZlan" matches[n] ".lan"`).
- `NSLocaleSensitivePredicateOption`: Indicates that strings to be compared using `<`, `<=`, `=`, `=>`, `>` should be handled in a locale-aware fashion. You represent this option in a predicate format string using a `[l]` following one of the `<`, `<=`, `=`, `=>`, `>` operators (for example, `"straße" >[l] "strasse"`).

Block Predicates

Finally, if you just can't be bothered to learn the `NSPredicate` format syntax, you can go through the motions with `NSPredicate +predicateWithBlock:`.

Swift Objective-C

```swift
let shortNamePredicate = NSPredicate { (evaluatedObject, _) in
    return (evaluatedObject as! Person).firstName.utf16.count <= 5
}

people.filtered(using: shortNamePredicate)
// ["Alice Smith", "Bob Jones"]
```

...Alright, that whole dig on `predicateWithBlock:` as being the lazy way out wasn't *entirely* charitable.

Actually, since blocks can encapsulate any kind of calculation, there is a whole class of queries that can't be expressed with the `NSPredicate` format string (such as evaluating against values dynamically calculated at run-time). And while its possible to accomplish the same using an `NSExpression` with a custom selector, blocks provide a convenient interface to get the job done.

One important note: **`NSPredicate`s created with `predicateWithBlock:` cannot be used for Core Data fetch requests backed by a `SQLite` store.**

`NSPredicate` is, and I know this is said a lot, truly one of the jewels of Cocoa. Other languages would be lucky to have something with half of its capabilities in a third-party library—let alone the standard library. Having it as a standard-issue component affords us as application and framework developers an incredible amount of leverage in working with data.

Together with `NSExpression`, `NSPredicate` reminds us what a treat Foundation is: a framework that is not only incredibly useful, but meticulously architected and engineered, to be taken as inspiration for how we should write our own code.