NSOperation Class Reference

Cocoa > Process Management

2008-11-19

Contents

NSOperation Class Reference

Inherits from	NSObject
Conforms to	NSObject (NSObject)
Framework	/System/Library/Frameworks/Foundation.framework
Availability	Available in Mac OS X v10.5 and later.
Companion guide	Threading Programming Guide
Declared in	NSOperation.h
Related sample code	NSOperationSample

Overview

The `NSOperation` class manages the execution of a single encapsulated task. Operations are typically scheduled by adding them to an operation queue object (an instance of the `NSOperationQueue` class), although you can also execute them directly by explicitly invoking their `start` method.

Operation objects are single-shot objects, that is, they perform their task once. You cannot reuse the same `NSOperation` object to perform a task (or a slight variant of the task) multiple times in succession. Attempting to execute an operation that has already finished results in an exception.

When manually executing operations, you are responsible for making sure the object is ready to execute. Starting an operation that is not in the ready state generally results in an exception being thrown. If you use an operation queue to manage the execution, the `NSOperationQueue` object ensures that the operation is executed only when it is ready.

Concurrent Versus Non-Concurrent Operations

Operation objects can be designed for either concurrent or non-concurrent operation. In the context of an `NSOperation` object, the terms concurrent and non-concurrent do not necessarily refer to the side-by-side execution of threads. Instead, a non-concurrent operation is one that executes using the environment that is provided for it while a concurrent operation is responsible for setting up its own execution environment. To understand how this might work in your code, look at the `NSOperationQueue` object as an example. For a non-concurrent operation, an operation queue automatically creates a thread and calls the operation object's `start` method, the default implementation of which configures the thread environment and calls the operation object's `main` method to run your custom code. For a concurrent operation, the queue simply calls the object's `start` method on the current thread. The operation object is then responsible for setting up the appropriate execution environment, which could include starting a new thread.

If you always design your operations to execute on a thread, then creating non-concurrent operations is the simplest way to go. There are some situations though where you might want to create a concurrent operation instead, including the following:

- You want to create the thread yourself.
- You want to launch a separate task instead of a thread.
- Your operation's main method initiates an asynchronous call and exits. (In such a situation, the callback function or method would then pass control to the operation object to process the request. For example, you could use this technique to set up a timer and then use the methods of the operation object to do some work each time the timer fires.)

By default, operations are designated as non-concurrent. For information on how to create a concurrent operation object, see the subclassing notes for this class.

Operation Dependencies

You can configure an operation to depend on the completion of other operations by adding those operations as dependencies. An operation object that has dependencies does not execute until all of its dependent operation objects finish executing. Once the last dependent operation finishes, the operation object moves to the ready state.

If a dependent operation is unable to perform its task for some reason, it is the responsibility of your code to make that determination. Operation objects that are non-concurrent (that is, their `isConcurrent` method returns `NO`) automatically catch and suppress any exceptions thrown by the operation object's `main` method. Thus, an operation that generates an exception may appear to finish normally even if it did not. If you need to track errors in a dependent operation, you must build that capability into the `main` method of your operation objects explicitly.

KVO-Compliant Properties

The `NSOperation` class is key-value coding (KVC) and key-value observing (KVO) compliant for several of its properties. As needed, you can observe these properties to control other parts of your application. The properties you can observe include the following:

- `isCancelled` - read-only property
- `isConcurrent` - read-only property
- `isExecuting` - read-only property
- `isFinished` - read-only property
- `isReady` - read-only property
- `dependencies` - read-only property
- `queuePriority` - readable and writable property

Although you can attach observers to these properties, you should not use Cocoa bindings to bind them to elements of your application's user interface. Code associated with your user interface typically must execute only in your application's main thread. Because an operation may execute in any thread, any KVO notifications associated with that operation may similarly occur in any thread.

If you override any of the preceding properties, your implementations must maintain KVC and KVO compliance. If you define additional properties for your `NSOperation` objects, it is recommended that you make those properties KVC and KVO compliant as well. For information on how to support key-value coding, see *Key-Value Coding Programming Guide*. For information on how to support key-value observing, see *Key-Value Observing Programming Guide*.

Threading Considerations

The methods of the `NSOperation` class implement automatic synchronization on the current instance. It is therefore safe to use a single instance of the `NSOperation` object from multiple threads without creating additional locks to synchronize access to the object.

When you subclass `NSOperation`, the methods in your implementation should also be safe to call from multiple threads. For example, if the methods of your operation object access shared resources, they should take the appropriate locks to synchronize access to those resources. For more information about writing thread-safe code, see *Threading Programming Guide*.

Subclassing Notes

The `NSOperation` class does not do anything by default and must be subclassed to perform any desired tasks. How you create your subclass depends on whether your operation is designed to execute concurrently or non-concurrently with respect to the thread that started the operation.

Methods to Override

For non-concurrent operations, you typically implement only one method:

- `main`

In your `main` method, you implement the code needed to perform the given operation. The `NSOperation` class manages the changes in state for your operation automatically and reports the appropriate condition of your operation from its methods.

If you are creating a concurrent operation, you need to override the following methods:

- `start`
- `isConcurrent`
- `isExecuting`
- `isFinished`

In your `start` method, you must prepare the operation for execution, which includes preparing the runtime environment for your operation. (For example, if you wanted to create a thread yourself, you would do it here.) Once your runtime environment is established, you can call any methods or functions you want to subsequently start your operation. Your implementation of the `start` method should not invoke `super`.

When implementing a concurrent operation, your custom subclass is responsible for reporting some of the state information associated with running the operation. In particular, you must override the `isExecuting` and `isFinished` methods to report on the current execution state of your operation. These methods must return accurate values for the state of your operation at all times, including when your operation has been cancelled. Your overridden methods should be KVO compliant.

Responding to the Cancel Command

An operation is responsible for periodically calling its own `isCancelled` method and aborting execution if it ever returns `YES`. Because it is bad form to kill a thread outright, the `NSOperationQueue` object sends a `cancel` message to your operation object if it ever needs your object to stop executing. (Other entities can similarly call the `cancel` method on an executing operation to ask it to stop.) The need to cancel an operation can typically arise from a user request or in a situation where the application or system is shutting down. When detected, your operation should clean up its environment and exit as soon as possible.

If an operation is cancelled, it should still update its internal state variables to reflect the change in execution status. In particular, the object's `isFinished` method should return `YES` and its `isExecuting` method should return `NO`. It must do this even if the it was cancelled before it started executing.

Note: If you implement a custom operation object as a concurrent operation, the `start` method can still be called even if the operation has already been cancelled. Your startup code should be prepared to handle this situation and clean up appropriately.

Tasks

Initialization

- `init` (page 11)
  Returns an initialized `NSOperation` object.

Executing the Operation

- `start` (page 15)
  Begins the execution of the operation.
- `main` (page 13)
  Performs the receiver's non-concurrent task.

Canceling Operations

- `cancel` (page 10)
  Advises the operation object that it should stop executing its task.

Getting the Operation Status

`– isCancelled` (page 11)
: Returns a Boolean value indicating whether the operation has been cancelled.

`– isExecuting` (page 12)
: Returns a Boolean value indicating whether the operation is currently executing.

`– isFinished` (page 12)
: Returns a Boolean value indicating whether the operation is done executing.

`– isConcurrent` (page 12)
: Returns a Boolean value indicating whether the operation runs asynchronously.

`– isReady` (page 13)
: Returns a Boolean value indicating whether the receiver's operation can be performed now.

Managing Dependencies

`– addDependency:` (page 9)
: Makes the receiver dependent on the completion of the specified operation.

`– removeDependency:` (page 14)
: Removes the receiver's dependence on the specified operation.

`– dependencies` (page 10)
: Returns a new array object containing the operations on which the receiver is dependent.

Prioritizing Operations in an Operation Queue

`– queuePriority` (page 14)
: Returns the priority of the operation in an operation queue.

`– setQueuePriority:` (page 15)
: Sets the priority of the operation when used in an operation queue.

Instance Methods

addDependency:

Makes the receiver dependent on the completion of the specified operation.

```
- (void)addDependency:(NSOperation *)operation
```

Parameters

operation
: The operation on which the receiver is dependent. The same dependency should not be added more than once to the receiver, and the results of doing so are undefined.

Discussion
The receiver is not considered ready to execute until all of its dependent operations finish executing. If the receiver is already executing its task, adding dependencies is unlikely to have any practical effect. This method may change the `isReady` and `dependencies` properties of the receiver.

It is a programmer error to create any circular dependencies among a set of operations. Doing so can cause a deadlock among the operations and may freeze your program.

Availability
Available in Mac OS X v10.5 and later.

See Also
`– removeDependency:` (page 14)
`– dependencies` (page 10)

Declared In
`NSOperation.h`

cancel

Advises the operation object that it should stop executing its task.

```
- (void)cancel
```

Discussion
This method does not force your operation code to stop. The code for your operation must invoke the `isCancelled` method periodically to determine whether the operation should be stopped. Once cancelled, an operation cannot be restarted.

If the operation is already finished executing, this method has no effect. Canceling an operation that is currently in an operation queue, but not yet executing, causes it to be removed from the queue (although not necessarily right away).

Availability
Available in Mac OS X v10.5 and later.

See Also
`– isCancelled` (page 11)

Declared In
`NSOperation.h`

dependencies

Returns a new array object containing the operations on which the receiver is dependent.

```
- (NSArray *)dependencies
```

Return Value
A new array object containing the `NSOperation` objects.

Discussion
The receiver is not considered ready to execute until all of its dependent operations finish executing.

Availability
Available in Mac OS X v10.5 and later.

See Also
- `addDependency:` (page 9)
- `removeDependency:` (page 14)

Declared In
`NSOperation.h`

init

Returns an initialized `NSOperation` object.

```
- (id)init
```

Return Value
The initialized `NSOperation` object.

Availability
Available in Mac OS X v10.5 and later.

Declared In
`NSOperation.h`

isCancelled

Returns a Boolean value indicating whether the operation has been cancelled.

```
- (BOOL)isCancelled
```

Return Value
`YES` if the operation was explicitly cancelled by an invocation of the receiver's `cancel` method; otherwise, `NO`. This method may return `YES` even if the operation is currently executing.

Discussion
Canceling an operation does not actively stop the receiver's code from executing. An operation object is responsible for calling this method periodically and stopping itself if the method returns `YES`.

Availability
Available in Mac OS X v10.5 and later.

See Also
- `cancel` (page 10)

Related Sample Code
NSOperationSample

Declared In
`NSOperation.h`

isConcurrent

Returns a Boolean value indicating whether the operation runs asynchronously.

```
- (BOOL)isConcurrent
```

Return Value
`YES` if the operation is asynchronous; otherwise, `NO` if the operation runs synchronously on whatever thread started it. This method returns `NO` by default.

Discussion
If you are implementing a concurrent operation, you must override this method and return `YES` from your implementation. For more information about the differences between concurrent and non-concurrent operations, see "Concurrent Versus Non-Concurrent Operations" (page 5).

Availability
Available in Mac OS X v10.5 and later.

Declared In
`NSOperation.h`

isExecuting

Returns a Boolean value indicating whether the operation is currently executing.

```
- (BOOL)isExecuting
```

Return Value
`YES` if the operation is executing; otherwise, `NO` if the operation has not been started or is already finished.

Discussion
If you are implementing a concurrent operation, you should override this method to return the execution state of your operation. Concurrent operations are also responsible for generating the appropriate KVO notifications whenever the execution state changes. For more information about manually generating KVO notifications, see *Key-Value Observing Programming Guide*.

Availability
Available in Mac OS X v10.5 and later.

Declared In
`NSOperation.h`

isFinished

Returns a Boolean value indicating whether the operation is done executing.

```
- (BOOL)isFinished
```

Return Value
`YES` if the operation is no longer executing; otherwise, `NO`.

Discussion
If you are implementing a concurrent operation, you should override this method to return the finished state of your operation. Concurrent operations are also responsible for generating the appropriate KVO notifications whenever the finished state changes. For more information about manually generating KVO notifications, see *Key-Value Observing Programming Guide*.

Availability
Available in Mac OS X v10.5 and later.

Declared In
`NSOperation.h`

isReady

Returns a Boolean value indicating whether the receiver's operation can be performed now.

```
- (BOOL)isReady
```

Return Value
YES if the operation can be performed now; otherwise, NO.

Discussion
Operations may not be ready due to dependencies on other operations or because of external conditions that might prevent needed data from being ready. The `NSOperation` class manages dependencies on other operations and reports the readiness of the receiver based on those dependencies.

Note: If the receiver is cancelled before it starts, operations that are dependent on the completion of the receiver will never become ready.

If your operation object has additional dependencies, you must override this method and return a value that accurately reflects the readiness of the receiver. Your custom implementation should invoke `super` and incorporate its return value into this method's return value. Your custom implementation must be KVO compliant.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- dependencies` (page 10)

Declared In
`NSOperation.h`

main

Performs the receiver's non-concurrent task.

```
- (void)main
```

Discussion
The default implementation of this method does nothing. For non-concurrent operations, you must override this method in your `NSOperation` subclass to perform the desired task. In your implementation, do not invoke `super`.

If you are implementing a concurrent operation, you should override the `start` method instead. In your overridden `start` method, you can continue to call this method to do the actual work if separating the work from your starting logic is practical.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- start` (page 15)

Declared In
`NSOperation.h`

queuePriority

Returns the priority of the operation in an operation queue.

```
- (NSOperationQueuePriority)queuePriority
```

Return Value
The relative priority of the operation. The returned value always corresponds to one of the predefined constants. (For a list of valid values, see "Operation Priorities" (page 16).) If no priority is explicitly set, this method returns `NSOperationQueuePriorityNormal`.

Availability
Available in Mac OS X v10.5 and later.

See Also
`- setQueuePriority:` (page 15)

Declared In
`NSOperation.h`

removeDependency:

Removes the receiver's dependence on the specified operation.

```
- (void)removeDependency:(NSOperation *)operation
```

Parameters

`operation`
The dependent operation to be removed from the receiver.

Discussion
This method may change the `isReady` and `dependencies` properties of the receiver.

Availability
Available in Mac OS X v10.5 and later.

See Also

- `addDependency:` (page 9)
- `dependencies` (page 10)

Declared In

`NSOperation.h`

setQueuePriority:

Sets the priority of the operation when used in an operation queue.

```
- (void)setQueuePriority:(NSOperationQueuePriority)priority
```

Parameters

`priority`

The relative priority of the operation. For a list of valid values, see "Operation Priorities" (page 16).

Discussion

You should use priority values only as needed to classify the relative priority of non-dependent operations. Priority values should not be used to implement dependency management among different operation objects. If you need to establish dependencies between operations, use the `addDependency:` method instead.

If you attempt to specify a priority value that does not match one of the defined constants, this method automatically adjusts the value you specify towards the `NSOperationQueuePriorityNormal` priority, stopping at the first valid constant value. For example, if you specified the value -10, this method would adjust that value to match the `NSOperationQueuePriorityVeryLow` constant. Similarly, if you specified +10, this method would adjust the value to match the `NSOperationQueuePriorityVeryHigh` constant.

Availability

Available in Mac OS X v10.5 and later.

See Also

- `queuePriority` (page 14)
- `addDependency:` (page 9)

Related Sample Code

NSOperationSample

Declared In

`NSOperation.h`

start

Begins the execution of the operation.

```
- (void)start
```

Discussion

The default implementation of this method configures the execution environment for a non-concurrent operation and invokes the receiver's main method. As part of the default configuration, this method performs several checks to ensure that the non-concurrent operation can actually run and generates appropriate KVO notifications for each change in the operation's state. If the receiver's operation has already been performed, this method throws an `NSInvalidArgumentException` exception. If the operation has already been cancelled, this method simply returns without calling main. If the operation is to be performed on a separate thread, this method may return before the operation itself completes on the other thread.

Note: An operation may not be ready to execute if it is dependent on other operations that have not yet finished.

If you are implementing a concurrent operation, you must override this method to initiate your operation; however, your implementation must not call `super`. If you override this method, you must also override the `isExecuting` and `isFinished` methods to report when your operation begins executing and finishes. Your implementations for these methods must maintain KVO compliance for the associated properties by manually sending the appropriate value change messages. For more information about manually generating KVO notifications, see *Key-Value Observing Programming Guide*.

Availability
Available in Mac OS X v10.5 and later.

See Also
`– main` (page 13)
`– isReady` (page 13)
`– dependencies` (page 10)

Declared In
`NSOperation.h`

Constants

NSOperationQueuePriority

Describes the priority of an operation relative to other operations in an operation queue.

```
typedef NSInteger NSOperationQueuePriority;
```

Availability
Available in Mac OS X v10.5 and later.

Declared In
`NSOperation.h`

Operation Priorities

These constants let you prioritize the order in which operations execute.

```
enum {
    NSOperationQueuePriorityVeryLow = -8,
    NSOperationQueuePriorityLow = -4,
    NSOperationQueuePriorityNormal = 0,
    NSOperationQueuePriorityHigh = 4,
    NSOperationQueuePriorityVeryHigh = 8
};
```

Constants

`NSOperationQueuePriorityVeryLow`

Operations receive very low priority for execution.

Available in Mac OS X v10.5 and later.

Declared in `NSOperation.h`.

`NSOperationQueuePriorityLow`

Operations receive low priority for execution.

Available in Mac OS X v10.5 and later.

Declared in `NSOperation.h`.

`NSOperationQueuePriorityNormal`

Operations receive the normal priority for execution.

Available in Mac OS X v10.5 and later.

Declared in `NSOperation.h`.

`NSOperationQueuePriorityHigh`

Operations receive high priority for execution.

Available in Mac OS X v10.5 and later.

Declared in `NSOperation.h`.

`NSOperationQueuePriorityVeryHigh`

Operations receive very high priority for execution.

Available in Mac OS X v10.5 and later.

Declared in `NSOperation.h`.

Discussion

You can use these constants to specify the relative ordering of operations that are waiting to be started in an operation queue. You should always use these constants (and not the defined value) for determining priority.

Declared In

`NSOperation.h`

Document Revision History

This table describes the changes to *NSOperation Class Reference*.

Date	Notes
2008-11-19	Updated the guidelines related to KVO compliance.
2008-10-15	Clarified cancellation semantics for concurrent operations.
2007-04-30	New document describing methods for managing encapsulated tasks.

Index