Core Data Model Versioning and Data Migration Programming Guide

Developer

Contents

lightweight migration（不需要提供Mapping Model；不需要自己控制version skew detection and migration bootstrapping; 不需要执行custom migration code)

NSMappingModel

包括default migration progress和custom migration progress.
二者的相同之处：均需要提供mapping model，entity migration

关于迁移的性能问题：
1. lightweight 是最佳选择。因为core data无需把数据加载到内存中来，迁移纯粹发生在sqlite内部
2. 对于default migration和custom migration来做，可以用multiple passes的方式来避免内存占用过大

Figures and Listings

Core Data Model Versioning and Data Migration

Core Data provides support for managing changes to a managed object model as your application evolves.

You can only open a Core Data store using the managed object model used to create it. Changing a model will therefore make it incompatible with (and so unable to open) the stores it previously created. If you change your model, you therefore need to change the data in existing stores to new version—changing the store format is known as *migration*.

To migrate a store, you need both the version of the model used to create it, and the current version of the model you want to migrate to. You can create a *versioned model* that contains more than one version of a managed object model. Within the versioned model you mark one version as being the current version. Core Data can then use this model to open persistent stores created using any of the model versions, and migrate the stores to the current version. To help Core Data perform the migration, though, you may have to provide information about how to map from one version of the model to another. This information may be in the form of hints within the versioned model itself, or in a separate mapping model file that you create.

At a Glance

Typically, as it evolves from one version to another, numerous aspects of your application change: the classes you implement, the user interface, the file format, and so on. You need to be aware of and in control of all these aspects; there is no API that solves the problems associated with all these—for example Cocoa does not provide a means to automatically update your user interface if you add a new attribute to an entity in your managed object model. Core Data does not solve all the issues of how you roll out your application. It does, though, provide support for a small—but important and non-trivial—subset of the tasks you must perform as your application evolves.

- Model versioning allows you to specify and distinguish between different configurations of your schema.

  There are two distinct views of versioning: your perspective as a developer, and Core Data's perspective. These may not always be the same. The differences are discussed in Understanding Versions (page 7).

  The format of a versioned managed object model, and how you add a version to a model, is discussed in Model File Format and Versions (page 10).

- Core Data needs to know how to map from the entities and properties in a source model to the entities and properties in the destination model.

In many cases, Core Data can infer the mapping from existing versions of the managed object model. This is described in Lightweight Migration (page 12).

If you make changes to your models such that Core Data cannot infer the mapping from source to destination, you need to create a mapping model. A mapping model parallels a managed object model, specifying how to transform objects in the source into instances appropriate for the destination.

How you create a mapping model is discussed in Mapping Overview (page 17).

- Data migration allows you to convert data from one model (schema) to another, using mappings.

  The migration process itself is discussed in The Migration Process (page 20).

  How you perform a migration is discussed in Initiating the Migration Process (page 23).

  You can also customize the migration process—that is, how you programmatically determine whether migration is necessary; how you find the correct source and destination models and the appropriate mapping model to initialize the migration manager; and then how you perform the migration.

  You only customize the migration process if you want to initiate migration yourself. You might do this to, for example, search locations other than the application's main bundle for models or to deal with large data sets by performing the migration in several passes using different mapping models.

  How you can customize the process is described in Customizing the Migration Process (page 26).

- If you are using iCloud, there are some constraints on what migration you can perform.

  If you are using iCloud, you must use lightweight migration. Other factors to be aware of are described in Migration and iCloud (page 30).

Although Core Data makes versioning and migration easier than would typically otherwise be the case, these processes are still non-trivial in effect. You still need to carefully consider the implications of releasing and supporting different versions of your application.

Prerequisites

This document assumes that you are familiar with the Core Data architecture and the fundamentals of using Core Data. You should be able to identify the parts of the Core Data stack and understand the roles of the model, the managed object context, and the persistent store coordinator. You need to know how to create a managed object model, how to create and programmatically interact with parts of the Core Data stack.

If you do not meet these requirements, you should first read the *Core Data Programming Guide* and related materials. You are strongly encouraged also to work through the *Core Data Utility Tutorial*.

Understanding Versions

There are two distinct views of versioning: your perspective as a developer, and Core Data's perspective. These may not always be the same—consider the following models.

Figure 1-1 Recipes models "Version 1.0"

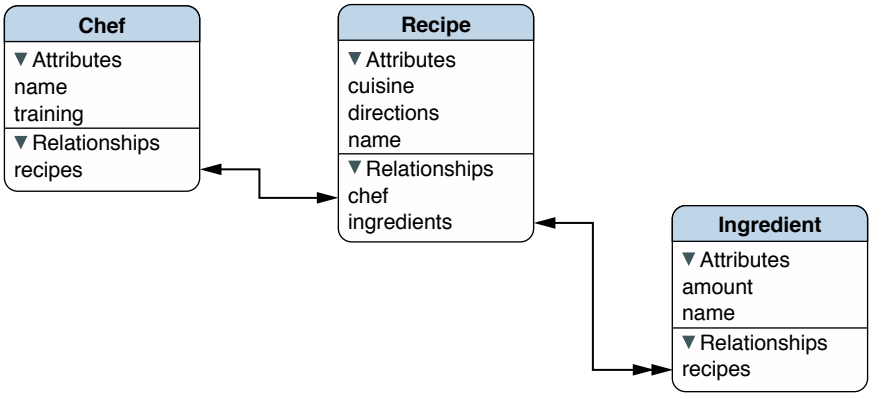


Figure 1-2 Recipes model "Version 1.1"

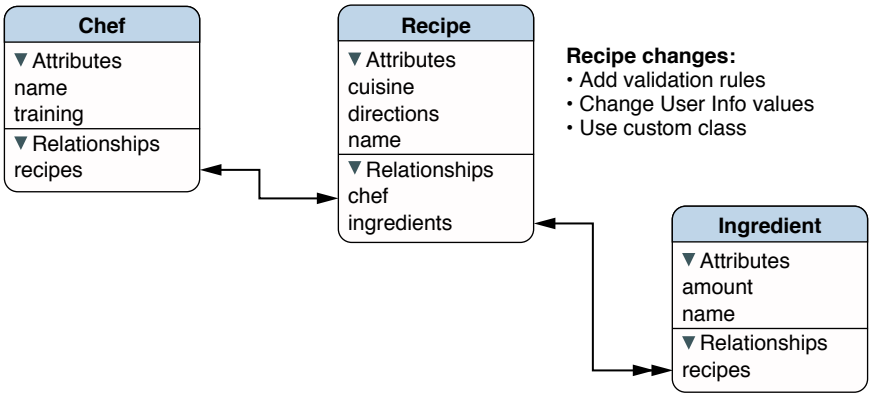
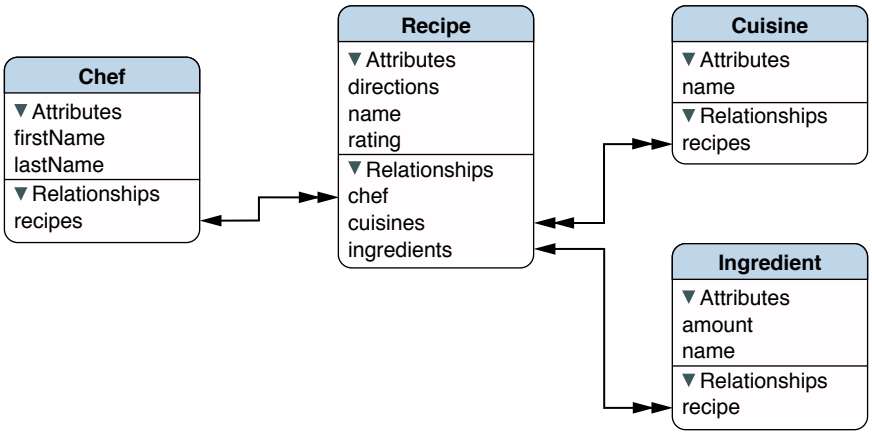


Figure 1-3 Recipes model "Version 2.0"



As a developer, your perspective is typically that a version is denoted by an identifier—a string or number, such as "9A218", "2.0.7", or "Version 1.1". To support this view, managed object models have a set of identifiers (see `versionIdentifiers`)—typically for a single model you provide a single string (the attribute itself is a set so that if models are merged all the identifiers can be preserved). How the identifier should be interpreted is up to you, whether it represents the version number of the application, the version that was committed prior to going on vacation, or the last submission before it stopped working.

Core Data, on the other hand, treats these identifiers simply as "hints". To understand why, recall that the format of a persistent store is dependent upon the model used to create it, and that to open a persistent store you must have a model that is compatible with that used to create it. Consider then what would happen if you changed the model but not the identifier—for example, if you kept the identifier the same but removed one entity and added two others. To Core Data, the change in the schema is significant, the fact that the identifier did *not* change is irrelevant.

Core Data's perspective on versioning is that it is only interested in features of the model that affect persistence. This means that for two models to be compatible:

- For each entity the following attributes must be equal: `name`, `parent`, `isAbstract`, and `properties`.

  `className`, `userInfo`, and validation predicates are not compared.

- For each property in each entity, the following attributes must be equal: `name`, `isOptional`, `isTransient`, `isReadOnly`, for attributes `attributeType`, and for relationships `destinationEntity`, `minCount`, `maxCount`, `deleteRule`, and `inverseRelationship`.

  `userInfo` and validation predicates are not compared.

Notice that Core Data ignores any identifiers you set. In the examples above, Core Data treats version 1.0 (Figure 1-1 (page 7)) and 1.1 (Figure 1-2 (page 7)) as being compatible.

Rather than enumerating through all the relevant parts of a model, Core Data creates a 32-byte hash digest of the components which it compares for equality (see `versionHash` (`NSEntityDescription`) and `versionHash` (`NSPropertyDescription`)). These hashes are included in a store's metadata so that Core Data can quickly determine whether the store format matches that of the managed object model it may use to try to open the store. (When you attempt to open a store using a given model, Core Data compares the version hashes of each of the entities in the store with those of the entities in the model, and if all are the same then the store is opened.) There is typically no reason for you to be interested in the value of a hash.

There may, however, be some situations in which you have two versions of a model that Core Data would normally treat as equivalent that you want to be recognized as being different. For example, you might change the name of the class used to represent an entity, or more subtly you might keep the model the same but change the internal format of an attribute such as a BLOB—this is irrelevant to Core Data, but it is crucial for the integrity of your data. To support this, Core Data allows you to set a hash modifier for an entity or property see `versionHashModifier` (`NSEntityDescription`) and `versionHashModifier` (`NSPropertyDescription`).

In the examples above, if you wanted to force Core Data to recognize that "Version 1.0" (Figure 1-1 (page 7)) and "Version 1.1" (Figure 1-2 (page 7)) of your models are different, you could set an entity modifier for the Recipe entity in the second model to change the version hash Core Data creates.

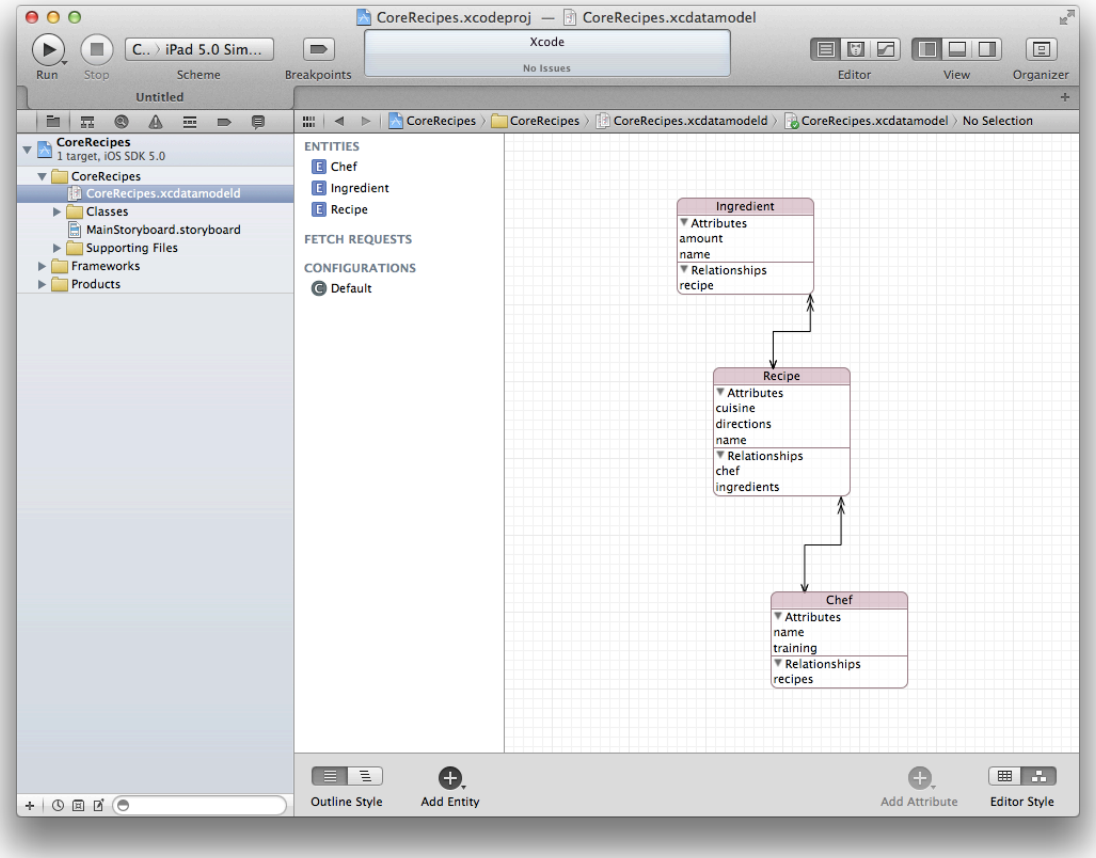
Model File Format and Versions

A managed object model that supports versioning is represented in the filesystem by a `.xcdatamodeld` document. An `.xcdatamodeld` document is a file package (see Document Packages) that groups versions of the model, each represented by an individual `.xcdatamodel` file, and an `Info.plist` file that contains the version information.

The model is compiled into a runtime format—a file package with a `.momd` extension that contains individually compiled model files with a `.mom` extension. You load the `.momd` model bundle using `NSManagedObjectModel`'s `initWithContentsOfURL:`.

To add a version to a model, you start with a model such as that illustrated in Figure 2-1.

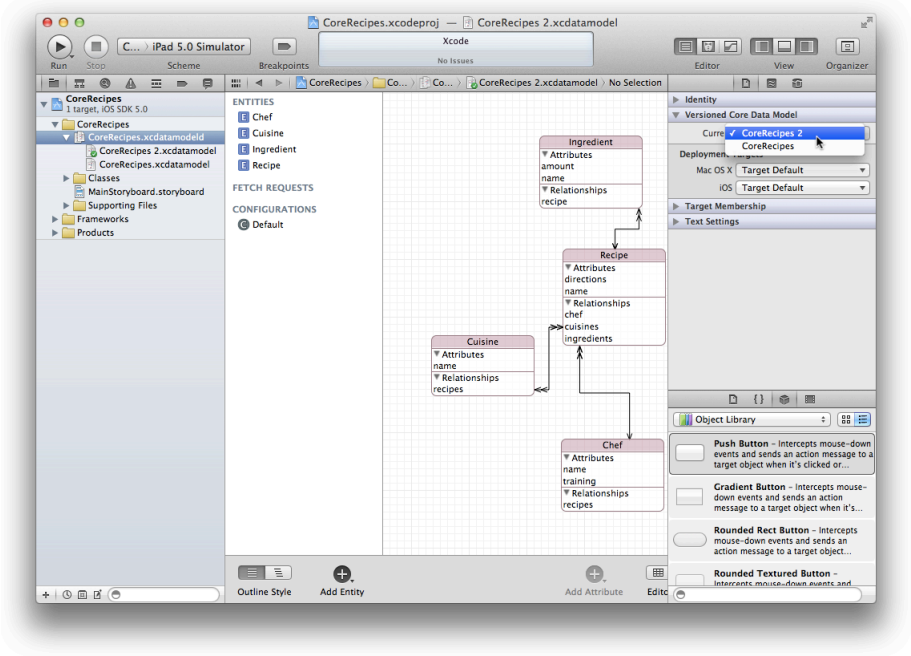
Figure 2-1 Initial version of the Core Recipes model



To add a version, select Editor > Add Model Version. In the sheet that appears, you enter the name of the new model version and select the model on which it should be based.

To set the new model as the current version of the model, select the `.xcdatamodeld` document in the project navigator, then select the new model in the pop-up menu in the Versioned Core Data Model area in the Attributes Inspector (see Figure 2-2).

Figure 2-2 Version 2 of the Core Recipes model



Lightweight Migration

If you just make simple changes to your model (such as adding a new attribute to an entity), Core Data can perform automatic data migration, referred to as *lightweight migration*. Lightweight migration is fundamentally the same as ordinary migration, except that instead of you providing a mapping model (as described in Mapping Overview (page 17)), Core Data infers one from differences between the source and destination managed object models.

Lightweight migration is especially convenient during early stages of application development, when you may be changing your managed object model frequently, but you don't want to have to keep regenerating test data. You can migrate existing data without having to create a custom mapping model for every model version used to create a store that would need to be migrated.

A further advantage of using lightweight migration—beyond the fact that you don't need to create the mapping model yourself—is that if you use an inferred model and you use the SQLite store, then Core Data can perform the migration in situ (solely by issuing SQL statements). This can represent a significant performance benefit as Core Data doesn't have to load any of your data. Because of this, you are encouraged to use inferred migration where possible, even if the mapping model you might create yourself would be trivial.

Core Data Must Be Able to Infer the Mapping

To perform automatic lightweight migration, Core Data needs to be able to find the source and destination managed object models itself at runtime. Core Data looks for models in the bundles returned by `NSBundle`'s `allBundles` and `allFrameworks` methods. If you store your models elsewhere, you must follow the steps described in Use a Migration Manager if Models Cannot Be Found Automatically (page 14). Core Data must then analyze the schema changes to persistent entities and properties and generate an inferred mapping model.

For Core Data to be able to generate an inferred mapping model, changes must fit an obvious migration pattern, for example:

- Simple addition of a new attribute
- Removal of an attribute
- A non-optional attribute becoming optional
- An optional attribute becoming non-optional, *and defining a default value*

- Renaming an entity or property

If you rename an entity or property, you can set the renaming identifier in the destination model to the name of the corresponding property or entity in the source model. You set the renaming identifier in the managed object model using the Xcode Data Modeling tool's property inspector (for either an entity or a property). For example, you can:

- Rename a Car entity to Automobile
- Rename a Car's `color` attribute to `paintColor`

The renaming identifier creates a "canonical name," so you should set the renaming identifier to the name of the property in the source model (unless that property already has a renaming identifier). This means you can rename a property in version 2 of a model then rename it again version 3, and the renaming will work correctly going from version 2 to version 3 or from version 1 to version 3.

In addition, Core Data supports:

- Adding relationships and changing the type of relationship
  - You can add a new relationship or delete an existing relationship.
  - Renaming a relationship (by using a renaming identifier, just like an attribute)
  - Changing a relationship from a to-one to a to-many, or a non-ordered to-many to ordered (and visa-versa)
- Changing the entity hierarchy
  - You can add, remove, rename entities
  - You can create a new parent or child entity and move properties up and down the entity hierarchy
  - You can move entities out of a hierarchy

    You *cannot* , however, merge entity hierarchies; if two existing entities do not share a common parent in the source, they cannot share a common parent in the destination

Request Automatic Migration Using an Options Dictionary

You request automatic lightweight migration using the options dictionary you pass in `addPersistentStoreWithType:configuration:URL:options:error:`, by setting values corresponding to both the `NSMigratePersistentStoresAutomaticallyOption` and the `NSInferMappingModelAutomaticallyOption` keys to `YES`:

```
NSError *error = nil;
```

```
NSURL *storeURL = <#The URL of a persistent store#>;
NSPersistentStoreCoordinator *psc = <#The coordinator#>;
NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithBool:YES], NSMigratePersistentStoresAutomaticallyOption,
    [NSNumber numberWithBool:YES], NSInferMappingModelAutomaticallyOption, nil];

BOOL success = [psc addPersistentStoreWithType:<#Store type#>
                    configuration:<#Configuration or nil#> URL:storeURL
                    options:options error:&error];
if (!success) {
    // Handle the error.
}
```

If you want to determine in advance whether Core Data can infer the mapping between the source and destination models without actually doing the work of migration, you can use `NSMappingModel`'s `inferredMappingModelForSourceModel:destinationModel:error:` method. This returns the inferred model if Core Data is able to create it, otherwise `nil`.

Use a Migration Manager if Models Cannot Be Found Automatically

To perform automatic migration, Core Data has to be able to find the source and destination managed object models itself at runtime (see Core Data Must Be Able to Infer the Mapping (page 12)). If you need to put your models in the locations not checked by automatic discovery, then you need to generate the inferred model and initiate the migration yourself using a migration manager (an instance of `NSMigrationManager`).

The following code sample illustrates how to generate an inferred model and initiate the migration using a migration manager. The code assumes that you have implemented two methods—`sourceModel` and `destinationModel`—that return the source and destination managed object models respectively.

```
- (BOOL)migrateStore:(NSURL *)storeURL toVersionTwoStore:(NSURL *)dstStoreURL
error:(NSError **)outError {


    // Try to get an inferred mapping model.
    NSMappingModel *mappingModel =
        [NSMappingModel inferredMappingModelForSourceModel:[self sourceModel]
                        destinationModel:[self destinationModel] error:outError];
```

```
    // If Core Data cannot create an inferred mapping model, return NO.
    if (!mappingModel) {
        return NO;
    }

    // Create a migration manager to perform the migration.
    NSMigrationManager *manager = [[NSMigrationManager alloc]
        initWithSourceModel:[self sourceModel] destinationModel:[self
destinationModel]];

    BOOL success = [manager migrateStoreFromURL:storeURL type:NSSQLiteStoreType
        options:nil withMappingModel:mappingModel toDestinationURL:dstStoreURL
        destinationType:NSSQLiteStoreType destinationOptions:nil error:outError];

    return success;
}
```

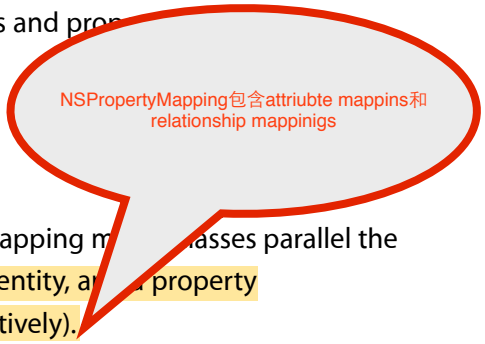
Note: Prior to OS X v10.7 and iOS 4, you need to use a store-specific migration manager to perform lightweight migration. You get the migration manager for a given persistent store type using `migrationManagerClass`, as illustrated in the following example.

```
- (BOOL)migrateStore:(NSURL *)storeURL toVersionTwoStore:(NSURL *)dstStoreURL
error:(NSError **)outError {


    // Try to get an inferred mapping model.
    NSMappingModel *mappingModel =
        [NSMappingModel inferredMappingModelForSourceModel:[self sourceModel]
                     destinationModel:[self destinationModel] error:outError];


    // If Core Data cannot create an inferred mapping model, return NO.
    if (!mappingModel) {
        return NO;
    }


    // Get the migration manager class to perform the migration.
    NSValue *classValue =
        [[NSPersistentStoreCoordinator registeredStoreTypes]
objectForKey:NSSQLiteStoreType];
    Class sqliteStoreClass = (Class)[classValue pointerValue];
    Class sqliteStoreMigrationManagerClass = [sqliteStoreClass
migrationManagerClass];


    NSMigrationManager *manager = [[sqliteStoreMigrationManagerClass alloc]
        initWithSourceModel:[self sourceModel] destinationModel:[self
destinationModel]];


    BOOL success = [manager migrateStoreFromURL:storeURL type:NSSQLiteStoreType
        options:nil withMappingModel:mappingModel toDestinationURL:dstStoreURL
       destinationType:NSSQLiteStoreType destinationOptions:nil error:outError];


    return success;
}
```

Mapping Overview

In many cases, Core Data may be able to infer how to transform data from one schema to another (see Lightweight Migration (page 12)). If Core Data cannot infer the mapping from one model to another, you need a definition of how to perform the transformation. This information is captured in a mapping model.

A *mapping model* is a collection of objects that specifies the transformations that are required to migrate part of a store from one version of your model to another (for example, that one entity is renamed, an attribute is added to another, and a third split into two). You typically create a mapping model in Xcode. Much as the managed object model editor allows you to graphically create the model, the mapping model editor allows you to customize the mappings between the source and destination entities and pro



Mapping Model Objects

Like a managed object model, a mapping model is a collection of objects. Mapping m asses parallel the managed object model classes—there are mapping classes for a model, an entity, a property (`NSMappingModel`, `NSEntityMapping`, and `NSPropertyMapping` respectively).

- An instance of `NSEntityMapping` specifies a source entity, a destination entity (the type of object to create to correspond to the source object) and mapping type (add, remove, copy as is, or transform).
- An instance of `NSPropertyMapping` specifies the name of the property in the source and in the destination entity, and a value expression to create the value for the destination property.

The model does not contain instances of `NSEntityMigrationPolicy` or any of its subclasses, however amongst other attributes instance of `NSEntityMapping` can specify the *name* of an entity migration policy class (a subclass of `NSEntityMigrationPolicy`) to use to customize the migration. For more about entity migration policy classes, see Custom Entity Migration Policies (page 21).

You can handle simple property migration changes by configuring a custom value expression on a property mapping directly in the mapping model editor in Xcode. For example, you can:

- Migrate data from one attribute to another.

  To rename `amount` to `totalCost`, enter the custom value expression for the `totalCost` property mapping as `$source.amount`.
- Apply a value transformation on a property.

To convert `temperature` from Fahrenheit to Celsius, use the custom value expression `($source.temperature - 32.0) / 1.8`.

- Migrate objects from one relationship to another.

  To rename `trades` to `transactions`, enter the custom value expression for the transactions property mapping as `FUNCTION($manager, "destinationInstancesForEntityMappingNamed:sourceInstances:", "TradeToTrade", $source.trades)`. (This assumes the entity mapping that migrates Trade instances is named TradeToTrade.)

There are six predefined keys you can reference in custom value expressions. To access these keys in source code, you use the constants as declared. To access them in custom value expression strings in the mapping model editor in Xcode, follow the syntax rules outlined in the predicate format string syntax guide and refer to them as:

`NSMigrationManagerKey`: `$manager`

`NSMigrationSourceObjectKey`: `$source`

`NSMigrationDestinationObjectKey`: `$destination`

`NSMigrationEntityMappingKey`: `$entityMapping`

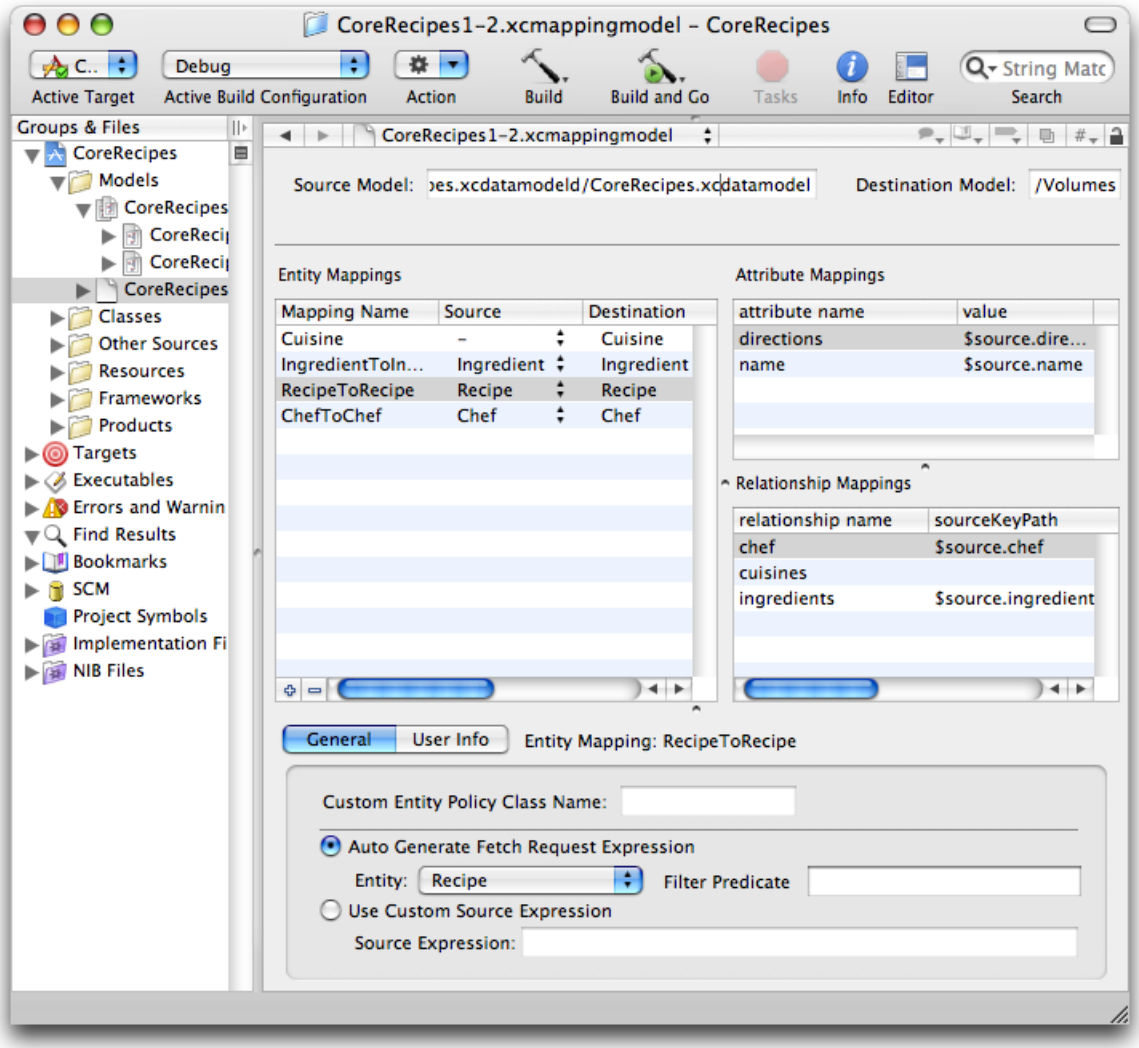
`NSMigrationPropertyMappingKey`: `$propertyMapping`

`NSMigrationEntityPolicyKey`: `$entityPolicy`

Creating a Mapping Model in Xcode

From the File menu, you select New File and in the New File pane select Design > Mapping Model. In the following pane, you select the source and destination models. When you click Finish, Xcode creates a new mapping model that contains as many default mappings as it can deduce from the source and destination. For example, given the model files shown in Figure 1-1 (page 7) and Figure 1-2 (page 7), Xcode creates a mapping model as shown in Figure 4-1.

Figure 4-1 Mapping model for versions 1-2 of the Core Recipes models



Reserved words in custom value expressions: If you use a custom value expression, you must escape reserved words such as `SIZE`, `FIRST`, and `LAST` using a `#` (for example, `$source.#size`).

The Migration Process

During migration, Core Data creates two stacks, one for the source store and one for the destination store. Core Data then fetches objects from the source stack and inserts the appropriate corresponding objects into the destination stack. Note that Core Data must *re-create* objects in the new stack.

Overview

Recall that stores are bound to their models. Migration is required when the model doesn't match the store. There are two areas where you get default functionality and hooks for customizing the default behavior:

- When detecting version skew and initializing the migration process.
- When performing the migration process.

To perform the migration process requires two Core Data stacks—which are automatically created for you—one for the source store, one for the destination store. The migration process is performed in 3 stages, copying objects from one stack to another.

Requirements for the Migration Process

Migration of a persistent store is performed by an instance of `NSMigrationManager`. To migrate a store, the migration manager requires several things:

- The managed object model for the destination store.

  This is the persistent store coordinator's model.
- A managed object model that it can use to open the existing store.
- Typically, a mapping model that defines a transformation from the source (the store's) model to the destination model.

  You don't need a mapping model if you're able to use lightweight migration—see Lightweight Migration (page 12).

You can specify custom entity migration policy classes to customize the migration of individual entities. You specify custom migration policy classes in the mapping model (note the "Custom Entity Policy Name" text field in Figure 4-1 (page 19)).

Custom Entity Migration Policies

If your new model simply adds properties or entities to your existing model, there may be no need to write any custom code. If the transformation is more complex, however, you might need to create a subclass of `NSEntityMigrationPolicy` to perform the transformation; for example:

- If you have a Person entity that also includes address information that you want to split into a separate Address entity, but you want to ensure uniqueness of each Address.
- If you have an attribute that encodes data in a string format that you want to change to a binary representation.

The methods you override in a custom migration policy correspond to the different phases of the migration process—these are called out in the description of the process given in Three-Stage Migration.

Three-Stage Migration

The migration process itself is in three stages. It uses a copy of the source and destination models in which the validation rules are disabled and the class of all entities is changed to `NSManagedObject`.

To perform the migration, Core Data sets up two stacks, one for the source store and one for the destination store. Core Data then processes each entity mapping in the mapping model in turn. It fetches objects of the current entity into the source stack, creates the corresponding objects in the destination stack, then recreates relationships between destination objects in a second stage, before finally applying validation constraints in the final stage.

Before a cycle starts, the entity migration policy responsible for the current entity is sent a `beginEntityMapping:manager:error:` message. You can override this method to perform any initialization the policy requires. The process then proceeds as follows:

1. Create destination instances based on source instances.

   At the beginning of this phase, the entity migration policy is sent a `createDestinationInstancesForSourceInstance:entityMapping:manager:error:` message; at the end it is sent a `endInstanceCreationForEntityMapping:manager:error:` message.

   In this stage, only attributes (not relationships) are set in the destination objects.

Instances of the source entity are fetched. For each instance, appropriate instances of the destination entity are created (typically there is only one) and their attributes populated (for trivial cases, `name = $source.name`). A record is kept of the instances per entity mapping since this may be useful in the second stage.

2. Recreate relationships.

   At the beginning of this phase, the entity migration policy is sent a `createRelationshipsForDestinationInstance:entityMapping:manager:error:` message; at the end it is sent a `endRelationshipCreationForEntityMapping:manager:error:` message.

   For each entity mapping (in order), for each destination instance created in the first step any relationships are recreated.

3. Validate and save.

   In this phase, the entity migration policy is sent a `performCustomValidationForEntityMapping:manager:error:` message.

   Validation rules in the destination model are applied to ensure data integrity and consistency, and then the store is saved.

At the end of the cycle, the entity migration policy is sent an `endEntityMapping:manager:error:` message. You can override this method to perform any clean-up the policy needs to do.

Note that Core Data cannot simply fetch objects into the source stack and insert them into the destination stack, the objects must be re-created in the new stack. Core Data maintains "association tables" which tell it which object in the destination store is the migrated version of which object in the source store, and vice-versa. Moreover, because it doesn't have a means to flush the contexts it is working with, you may accumulate many objects in the migration manager as the migration progresses. If this presents a significant memory overhead and hence gives rise to performance problems, you can customize the process as described in Multiple Passes—Dealing With Large Datasets (page 29).

Initiating the Migration Process

This chapter describes how to initiate the migration process and how the default migration process works. It does not describe customizing the migration process—this is described in Customizing the Migration Process (page 26).

Initiating the Migration Process

When you initialize a persistent store coordinator, you assign to it a managed object model (see `initWithManagedObjectModel:`); the coordinator uses that model to open persistent stores. You open a persistent store using `addPersistentStoreWithType:configuration:URL:options:error:`. How you use this method, however, depends on whether your application uses model versioning and on how you choose to support migration—whether you choose to use the default migration process or custom version skew detection and migration bootstrapping. The following list describes different scenarios and what you should do in each:

- Your application does not support versioning

  You use `addPersistentStoreWithType:configuration:URL:options:error:` directly.

  If for some reason the coordinator's model is not compatible with the store schema (that is, the version hashes current model's entities do not equal those in the store's metadata), the coordinator detects this, generates an error, and `addPersistentStoreWithType:configuration:URL:options:error:` returns `NO`. You must deal with this error appropriately.

- Your application does support versioning and you choose to use either the lightweight or the default migration process

  You use `addPersistentStoreWithType:configuration:URL:options:error:` as described in Lightweight Migration (page 12) and The Default Migration Process (page 24) respectively.

  The fundamental difference from the non-versioned approach is that you instruct the coordinator to automatically migrate the store to the current model version by adding an entry to the options dictionary where the key is `NSMigratePersistentStoresAutomaticallyOption` and the value is an `NSNumber` object that represents `YES`.

- Your application does support versioning and you choose to use custom version skew detection and migration bootstrapping

Before opening a store you use `isConfiguration:compatibleWithStoreMetadata:` to check whether its schema is compatible with the coordinator's model:

- If it is, you use `addPersistentStoreWithType:configuration:URL:options:error:` to open the store directly;
- If it is not, you must migrate the store first then open it (again using `addPersistentStoreWithType:configuration:URL:options:error:`).

You could simply use `addPersistentStoreWithType:configuration:URL:options:error:` to check whether migration is required, however this is a heavyweight operation and inefficient for this purpose.

1. 可在迁移过程中执行自定制的代码
2. 可以自定制版本检测和迁移引导程序

It is important to realize that there are two *orthogonal* concepts:

1. You can execute custom code during the migration.
2. You can have custom code for version skew detection and migration bootstrapping.

The migration policy classes allow you to customize the migration of entities and properties in a number of ways, and these are typically all you need. You might, however, use custom skew detection and migration bootstrapping so that you can take control of the migration process. For example, if you have very large stores you could set up a migration manager with the two data models, and then use a series of mapping models to migrate your data into your destination store (if you use the same destination URL for each invocation, Core Data adds new objects to the existing store). This allows the framework (and you) to limit the amount of data in memory during the conversion process.

default migration 和 custom migration的唯一区别

The Default Migration Process

To open a store and perform migration (if necessary), you use `addPersistentStoreWithType:configuration:URL:options:error:` and add to the options dictionary an entry where the key is `NSMigratePersistentStoresAutomaticallyOption` and the value is an `NSNumber` object that represents `YES`. Your code looks similar to the following example:

Listing 6-1 Opening a store using automatic migration

```
NSError *error;
NSPersistentStoreCoordinator *psc = <#The coordinator#>;
NSURL *storeURL = <#The URL of a persistent store#>;
NSDictionary *optionsDictionary =
    [NSDictionary dictionaryWithObject:[NSNumber numberWithBool:YES]
```

```
                    forKey:NSMigratePersistentStoresAutomaticallyOption];

NSPersistentStore *store = [psc addPersistentStoreWithType:<#Store type#>
                                   configuration:<#Configuration or nil#>
                                   URL:storeURL
                                   options:optionsDictionary
                                   error:&error];
```

If the migration proceeds successfully, the existing store at `storeURL` is renamed with a "~" suffix before any file extension and the migrated store saved to `storeURL`.

In its implementation of `addPersistentStoreWithType:configuration:URL:options:error:` Core Data does the following:

1. Tries to find a managed object model that it can use to open the store.

   Core Data searches through your application's resources for models and tests each in turn. If it cannot find a suitable model, Core Data returns `nil` and a suitable error.

2. Tries to find a mapping model that maps from the managed object model for the existing store to that in use by the persistent store coordinator.

   Core Data searches through your application's resources for available mapping models and tests each in turn. If it cannot find a suitable mapping, Core Data returns `NO` and a suitable error.

   Note that you must have created a suitable mapping model in order for this phase to succeed.

3. Creates instances of the migration policy objects required by the mapping model.

Note that even if you use the default migration process you can customize the migration itself using custom migration policy classes.

Customizing the Migration Process

You only customize the migration process if you want to initiate migration yourself. You might do this to, for example, to search for models in locations other than the application's main bundle, or to deal with large data sets by performing the migration in several passes using different mapping models (see Multiple Passes—Dealing With Large Datasets (page 29)).

Is Migration Necessary

Before you initiate a migration process, you should first determine whether it is necessary. You can check with `NSManagedObjectModel`'s `isConfiguration:compatibleWithStoreMetadata:` as illustrated in Listing 7-1 (page 26).

Listing 7-1 Checking whether migration is necessary

```
NSPersistentStoreCoordinator *psc = /* get a coordinator */ ;
NSString *sourceStoreType = /* type for the source store, or nil if not known */
;
NSURL *sourceStoreURL = /* URL for the source store */ ;
NSError *error = nil;

NSDictionary *sourceMetadata =
    [NSPersistentStoreCoordinator metadataForPersistentStoreOfType:sourceStoreType
                                URL:sourceStoreURL
                                error:&error];

if (sourceMetadata == nil) {
    // deal with error
}

NSString *configuration = /* name of configuration, or nil */ ;
NSManagedObjectModel *destinationModel = [psc managedObjectModel];
BOOL pscCompatibile = [destinationModel
```

```
            isConfiguration:configuration
            compatibleWithStoreMetadata:sourceMetadata];

if (pscCompatibile) {
    // no need to migrate
}
```

Initializing a Migration Manager

You initialize a migration manager using `initWithSourceModel:destinationModel:`; you therefore first need to find the appropriate model for the store. You get the model for the store using `NSManagedObjectModel`'s `mergedModelFromBundles:forStoreMetadata:`. If this returns a suitable model, you can create the migration manager as illustrated in Listing 7-2 (page 27) (this code fragment continues from Listing 7-1 (page 26)).

Listing 7-2 Initializing a Migration Manager

```
NSArray *bundlesForSourceModel = /* an array of bundles, or nil for the main bundle
 */ ;
NSManagedObjectModel *sourceModel =
    [NSManagedObjectModel mergedModelFromBundles:bundlesForSourceModel
                            forStoreMetadata:sourceMetadata];

if (sourceModel == nil) {
    // deal with error
}

MyMigrationManager *migrationManager =
    [[MyMigrationManager alloc]
            initWithSourceModel:sourceModel
            destinationModel:destinationModel];
```

Performing a Migration

You migrate a store using `NSMigrationManager`'s `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:`. To use this method you need to marshal a number of parameters; most are straightforward, the only one that requires some work is the discovery of the appropriate mapping model (which you can retrieve using `NSMappingModel`'s `mappingModelFromBundles:forSourceModel:destinationModel:` method). This is illustrated in Listing 7-3 (page 28) (a continuation of the example shown in Listing 7-2 (page 27)).

Listing 7-3 Performing a Migration

```
NSArray *bundlesForMappingModel = /* an array of bundles, or nil for the main
bundle */ ;
NSError *error = nil;


NSMappingModel *mappingModel =
    [NSMappingModel
            mappingModelFromBundles:bundlesForMappingModel
            forSourceModel:sourceModel
            destinationModel:destinationModel];


if (mappingModel == nil) {
        // deal with the error
}

NSDictionary *sourceStoreOptions = /* options for the source store */ ;
NSURL *destinationStoreURL = /* URL for the destination store */ ;
NSString *destinationStoreType = /* type for the destination store */ ;
NSDictionary *destinationStoreOptions = /* options for the destination store */ ;

BOOL ok = [migrationManager migrateStoreFromURL:sourceStoreURL
                  type:sourceStoreType
                  options:sourceStoreOptions
                  withMappingModel:mappingModel
                  toDestinationURL:destinationStoreURL
                  destinationType:destinationStoreType
                  destinationOptions:destinationStoreOptions
```

```
                    error:&error];
```

Multiple Passes—Dealing With Large Datasets

The basic approach shown above is to have the migration manager take two models, and then iterate over the steps (mappings) provided in a mapping model to move the data from one side to the next. Because Core Data performs a "three stage" migration—where it creates all of the data first, and then relates the data in a second stage—it must maintain “association tables" (which tell it which object in the destination store is the migrated version of which object in the source store, and vice-versa). Further, because it doesn't have a means to flush the contexts it is working with, it means you'll accumulate many objects in the migration manager as the migration progresses.

In order to address this, the mapping model is given as a parameter of the `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:` call itself. What this means is that if you can segregate parts of your graph (as far as mappings are concerned) and create them in separate mapping models, you could do the following:

1. Get the source and destination data models
2. Create a migration manager with them
3. Find all of your mapping models, and put them into an array (in some defined order, if necessary)
4. Loop through the array, and call `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:` with each of the mappings

This allows you to migrate "chunks" of data at a time, while not pulling in all of the data at once.

From a "tracking/showing progress” point of view, that basically just creates another layer to work from, so you'd be able to determine percentage complete based on number of mapping models to iterate through (and then further on the number of entity mappings in a model you've already gone through).

Migration and iCloud

If you are using iCloud, you can only migrate the contents of a store using automatic lightweight migration. To migrate a persistent store that is in iCloud, you add the store to a persistent store coordinator using `addPersistentStoreWithType:configuration:URL:options:error:` and pass at least the following options in the options dictionary:

```
NSDictionary *optionsDictionary = [[NSDictionary alloc] initWithObjectsAndKeys:
    [NSNumber numberWithBool:YES], NSInferMappingModelAutomaticallyOption,
    [NSNumber numberWithBool:YES], NSMigratePersistentStoresAutomaticallyOption,
    <#Ubiquitous content name#>, NSPersistentStoreUbiquitousContentNameKey, nil];
```

Changes to a store are recorded and preserved independently for each model version that is associated with a given `NSPersistentStoreUbiquitousContentNameKey`. A persistent store configured with a given `NSPersistentStoreUbiquitousContentNameKey` only syncs data with a store on another device data if the model versions match.

If you migrate a persistent store configured with a `NSPersistentStoreUbiquitousContentNameKey` option to a new model version, the store's history of changes originating from the current device will also be migrated and then merged with any other devices configured with that new model version. Any changes from stores using the new version are also merged in. Existing changes can *not*, however, be migrated to a new model version if the migration is performed using a custom mapping model.

Document Revision History

This table describes the changes to *Core Data Model Versioning and Data Migration Programming Guide*.

Date	Notes
2012-01-09	Updated to describe use of migration with iCloud.
2010-02-24	Added further details to the section on Mapping Model Objects.
2009-06-04	Added an article to describe the lightweight migration feature.
2009-03-05	First version for iOS.
2008-02-08	Added a note about migrating stores from OS X v10.4 (Tiger).
2007-05-18	New document that describes managed object model versioning and Core Data migration.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010