# Core Data 编程指南

发布于：2010-11-26 14:42 阅读数：21796

“一、技术概览 1. Core Data 功能初窥 对于处理诸如对象生命周期管理、对象图管理等日常任务，Core Data框架提供了广泛且自动化的解决方案。它有以下特性。 （注：对象图 – Object graph的解”



## 一、技术概览

**1. Core Data 功能初窥**

对于处理诸如对象生命周期管理、对象图管理等日常任务，Core Data框架提供了广泛且自动化的解决方案。它有以下特性。

（注：对象图 – Object graph的解释：在面向对象编程中，对象之间有各种关系，例如对象直接引用另外的对象，或是通过引用链间接的引用其他对象，这些关系组成了网状的结构。我们把这些对象（和它们之间的联系）成为对象图。 对象图可大可小，有繁有简。 只包含单个字符串对象的数组就是一个简单的代表；而包含了application对象，引用windows， menus和相关视图对象、其他对象这样的结构就是复杂对象图的例子——这是在说mainwindow.xib。

有时，你可能想要把这样的对象图转化形式，让它们可以被保存到文件中，以使其他的进程或其他的机器可以再次将保存的内容读出，重购对象。 这样的过程常被成之为“归档”(Archiving)。

有些对象图是不完整的——通常称之为局部对象图（partial object graphs）。局部对象图包含了“占位符”（Placeholder）对象，所谓”占位符“，就是一些暂时无内容的对象，它们将再后期被具体化。一个典型的例子就是nib文件中包含的File's Owner对象。

1) 对于key-value coding 和key-value observing完整且自动化的支持

除了为属性整合KVC和KVO的访问方法外， Core Data还整合了适当的集合访问方法来处理多值关系。

2) 自动验证属性(property)值

Core Data中的managed object扩展了标准的KVC 验证方法，以保证单个的数值在可接受的范围之内，从而使组合的值有意义。（需校准翻译）

3) 支持跟踪修改和撤销操作

对于撤销和重做的功能，除过用基本的文本编辑外，Core Data还提供内置的管理方式。

4) 关系的维护

Core Data管理数据的变化传播，包括维护对象间关系的一致性。

5) 在内存中和界面上分组、过滤、组织数据

6) 自动支持对象存储在外部数据仓库的功能

7) 创建复杂请求

你不需要动手去写复杂的SQL语句，就可以创建复杂的数据请求。方法是在“获取请求”(fetch request)中关联NSPredicate（又看到这个东东了，之前用它做过正则）。NSPrdicate支持基本的功能、相关子查询和其他高级的 SQL特性。它还支持正确的Unicode编码（不太懂，请高人指点）， 区域感知查询（据说就是根据区域、语言设置调整查询的行为）、排序和正则表达式。

8) 延迟操作（原文为Futures(faulting）直译为期货，这里个人感觉就是延迟操作的形象说法。请高人指教）。

Core Data 使用延迟加载(lazy loading)的方式减少内存负载。 它还支持部分实体化延迟加载，和“写时拷贝”的数据共享机制。（写时拷贝，说的是在复制对象的时候，实际上不生成新的空间，而是让对象共享一块存储区域，在其内容发生改变的时候再分配）。

9) 合并的策略

Core Data 内置了版本跟踪和乐观锁定(optimistic locking)来支持多用户写入冲突的解决。

注：乐观锁，假定数据一般不出现冲突，所以在数据提交更新的时候，才对数据的冲突进行检测，如果冲突了，就返回冲突信息。

10) 数据迁移

就开发工作和运行时资源来说，处理数据库架构的改变总是很复杂。Core Data的schema migration工具可以简化应对数据库结构变化的任务， 而且在某些情况下，允许你执行高效率的数据库原地迁移工作。

11) 可选择针对程序Controller层的集成，来支持UI的显示同步

Core Data在iPhone OS之上 提供NSFetchedResultsController对象来做相关工作，在Mac OS X上，我们用Cocoa提供的绑定(Binding)机制来完成。

**2. 为何要使用Core Data**

使用Core Data有很多原因，其中最简单的一条就是：它能让你为Model层写的代码的行数减少为原来的50%到70%。 这归功于之前提到的Core Data的特性。更妙的是，对于上述特性你也既不用去测试，也不用花功夫去优化。

Core Data拥有成熟的代码，这些代码通过单元测试来保证品质。应用Core Data的程序每天被世界上几百万用户使用。通过了几个版本的发布，已经被高度优化。 它能利用Model层的信息和运行时的特性，而不通过程序层的代码实现。 除了提供强大的安全支持和错误处理外，它还提供了最优的内存扩展性，可实现有竞争力的解决方案。不使用Core Data的话，你需要花很长时间来起草自己的方案，解决各种问题，这样做效率不高。

除了Core Data本身的优点之外，使用它还有其他的好处： 它很容易和Mac OS X系统的Tool chain集成；利用Model设计工具可以按图形化方式轻松创建数据库的结构；你可以用Instruments的相关模板来测试Core Data的效率并debug。 在Mac OS X的桌面程序中，Core Data还和Interface Builder集成（打开Inspector可以看到有binding的选项，这个东东iPhone上木有。。。），按照model来创建UI变的更简单了。 这些功能能更进一步的帮助你缩短设计、开发、测试程序的周期。

**3. Core Data不是。。。**

看了前面的介绍之后，我们还需要了解一下关于Core Data常见的误解：

1) Core Data不是一个关系型数据库，也不是关系型数据库管理系统(RDBMS)。

Core Data 为数据变更管理、对象存储、对象读取恢复的功能提供了支持。 它可以使用SQLite作为持久化存储的类型。 它本身并不是一个数据库（这点很重要，比如，你可以使用Core Data来记录数据变更，管理数据，但并不能用它向文件内存储数据）。

2) Core Data不是银弹

它并不能取代你写代码的工作。虽然可以纯粹使用XCode的数据建模工具和Interface Builder来编写复杂程序，但在更多的程序中，你都自己动手写代码。

3) Core Data并不依赖于Cocoa Bindings

Core Data + Cocoa Binding = 减少代码数量。但Core Data完全可以在没有bindings的条件下使用。例如，可以编写一个没有UI，但包含Core Data的程序。

## 二、Core Data基础

**1. Core Data基本架构**

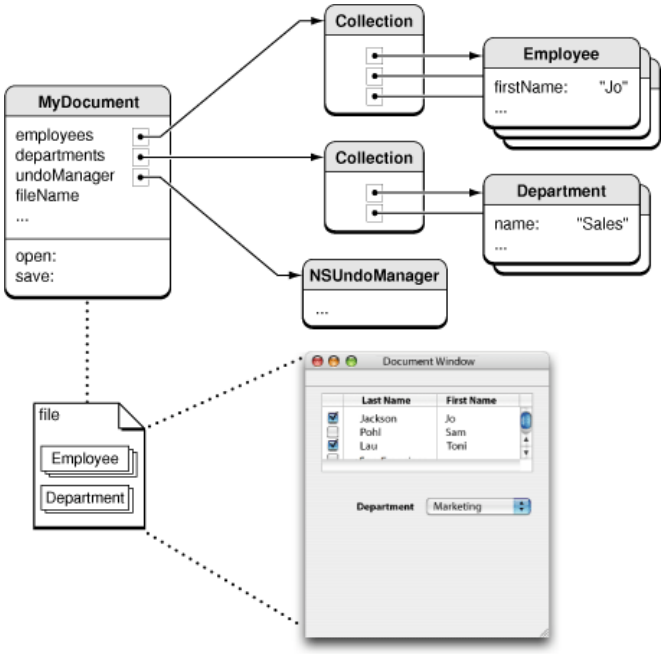
在大部分程序中，你要能通过某种方式打开一个包含对象归档的文件， 这个文件内至少要有一个根对象的引用。另外，还得能将所有的对象归档到文件中，如果你想要实现撤销的功能，就还要记录对象的更改情况。例如，在 Employee的示例程序中，你要能打开一个包含有employee和department对象归档的文件，而且这个文件至少包含了一个根对象——这 里，是一个包含所有employee的数组——请参考例图Figure 1。 相应的，你还要能将程序中的employee、department对象归档到文件中去。

Figure 1 按照Core Data文档结构管理的对象示意图


多线程编程指南【中文完整翻译版】


关于iCloud的注册，到代码的实现


iPhone消息推送机制实现与探讨




使用Core Data的框架，大多数的功能都可以自动实现，因为我们有managed object context（管理对象的上下文，有时直接叫"Context"）。managed object context就像是一个关卡，通过它可以访问框架底层的对象——这些对象的集合我们称之为"persistence stack"（数据持久栈）。 managed object context作为程序中对象和外部的数据存储的中转站。栈的底部是persistence object stores（持久化数据存储），请看Figure 2的示意图。

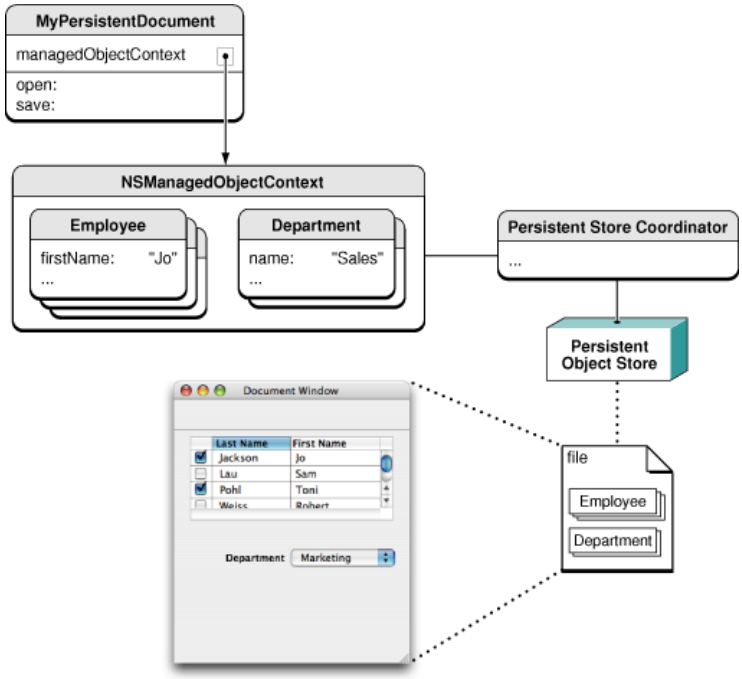



Figure 2 使用Core Data的文档管理示意图

Core Data的使用并不限制在基于文档的程序中（document-based application）。你也能创建一个包含Core Data 的Utility程序（请查看Core Data Utility tutorial文档）。当然其他类型的程序也都可以使用Core Data。

**被管理对象和上下文(Managed Objects and Contexts)**

你可以把被管理对象上下文想象成一个"聪明"的便笺簿。当你从数据持久层获取对象时，就把这些临时的数据拷贝拿到写在自己的便笺簿上（当然，在便笺上对象会 "恢复"以前的对象图结构）。然后你就可以随心所欲的修改这些值了（本子是你的，随便画都可以），除非你保存这些数据变化，否则持久层的东西是不会变 的。（跟修改文件后要保存是一个道理）。

附在Core Data框架中模型对象（Model objects）常被称为"被管理对象"(Managed objects)。所有的被管理对象都要通过上下文进行注册。使用上下文，你可以在对象图中添加、删除对象，并记录对象的更改（包括单个对象，或是对象间 的关系）。记录更改后就能支持撤销和重做的功能。同时，上下文还能保证关系更改后对象图的完整性。

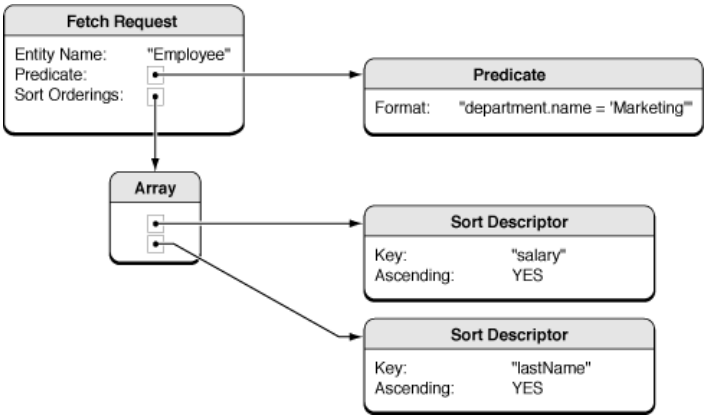
如果你想要保存所做的修改， 上下文会保证对象的有效性。在验证有效性后，更改会被写入到persistent s

tore(持久化存储层)中。你在程序中的添加和删除动作都会被作用在存储的数据中。

在你的一个程序中，可能存在多个上下文。 对于数据存储(store)中的每个对象，对应的都有唯一的一个被管理对象(managed object)和上下文相关联（详情请查看"Faulting and Uniquing"文档）。换个角度来想，在persistent store中存储的对象有可能被用在不同的上下文中，每个上下文都有与之对应的被管理对象，被管理对象可以被独立的修改，这样就可能在存储 时导致数据的不一致。Core Data提供了许多解决这个问题的途径（请查看"Using Managed Object"一章）。

**获取数据的请求(Fetch Requests)**

要使用上下文来获取数据，你需要创建相应的请求(Fetch request)。 Fetch request对象包含你想获取的对象的描述。例如：“所有 Employee”，或“所有的Employee，department是marketing，按薪资降序排列”。Fetch Request包含三个部分。使用最简单的写法，必须指定实体（Entity）的名称，这就暗示了，每次智能获得一种类型的实体。 Fetch Request 还可以包含谓词（predicate）——注：有些地方也把这个叫断言，个人感觉谓词更准确些。谓词将描述对象需要满足的条件（这就和我们在SQL里加的 限定条件差不多，正如前面的"All Employees, in the Marketing department"）。另外，Fetch Request还可包含一个用于描述排序方式的对象（熟悉的Order by操作）。如图Figure3所示：




在程序中，你将Fetch Request这个请求发送给上下文，上下文就会从相关的数据源中查找复合条件的对象（也可能找不到），并返回。 所有的被管理对象（managed object）都必须在上下文中注册，因此通过fetch request获得的对象自动被注册。但如前所述，每个在持久存储层（persistence store）中的对象都对应一个和上下文相关的被管理对象(managed object)，因此，如果在上下文中已经存在了fetch request要取的对象，那么这个被管理对象将被返回。

Core Data追求高执行效率。 它是“需求驱动”的，因此只会创建你确实需要的对象。对象图不需要保留所有在数据存储层中的对象。单纯指定数据持久层的动作不会将其中所有的数据放到上下 文中去。 当你想从数据存储层中获取某些对象的时候，你只会得到那些你请求的（有点罗嗦，总的意思就是需要时获取，获取的就是需要的）。如果你不在需要这个对象的时 候，默认情况下它会被释放。（当然，只是释放这个对象，而不是从对象图中移除该对象）。——注：个人感觉有点像重新拷了一个文件的某些部分，不用了就在副 本中删除，不会影响原件。

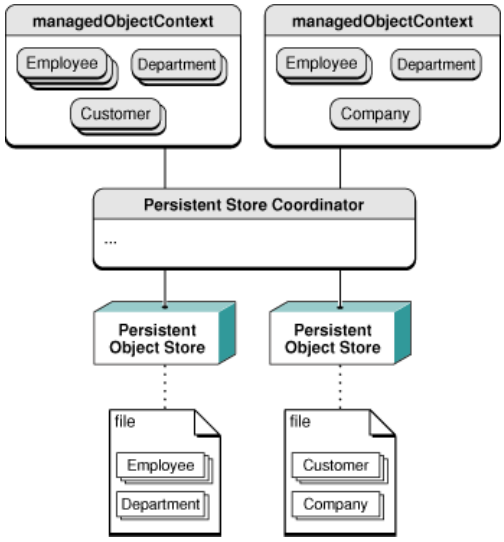
**持久化存储助理(Persistent Store Coordinator)**

之前提到过，程序中的对 象和外部存储的数据通过Core Data框架中的一系列对象进行协调，这一系列的对象总的被称为持久存储栈(Persistence stack)。在栈顶是被管理对象上下文(Managed object context)，而栈底是持久化对象存储层(Persistence object store)。在它们之间就是持久化存储助理。

事实上，持久化存储助理定义了一个栈。从设计方面考虑，它就是可以作为上下 文的”外观“， 这样多个数据存储(Persistence store)看起来就像是一个。 然后上下文就可以根据这些数据存储来创建对象图了。持久化存储助理智能关联一个被管理对象的模型。如果你像要把不同的实体放到不同的存储中去，就需要为你 的模型实体做“分区”，方式是通过定义被管理对象模型的configurations。（请参考"Configurations"一章）。

Figure 4演示了这样的一个结构：employees和departments存储在一个文件中，customers和companies存储在另外一个文件中。当你要获取对象的时候，它们从相关的文件中自动获取；当保存时，又被归档到相应的文件中。

Figure 4存储栈—改




**持久化存储(Persistent Stores)**

持久化存储是和单独的一个文件或外部的数据关联的，它负责将数据和上下文中的对象进行对应。通常，需要你直接和持久化对象存储打交道的地方，就是指定新的、 和程序进行关联的外部数据的位置（例如，当用户打开或保存一个文档）。大多数需要访问持久化存储的动作都由上下文来完成。

程序的代码——特别是和被管理对象相关的部分——不应该对持久化存储做任何假设（也就是不需要自己考虑存储的方式或过程）。 Core Data对几种文件格式有原生的支持。你可以选择一种自己程序需要的。假设在某个阶段你决定换一种文件的格式，而又不想修改程序的框架，而且，你的程序做 了适当的抽象（注：这个就属于设计方面的东东了），这时，你就能尝到使用Core Data的甜头了。例如，在最初的设计中，程序只从本地文件中获取数据，而你的程序没有去硬指定对应数据的获取位置，而是可以在后期指定从远程位置添加新 的数据类型，这样你就可以使用新的类型，而不需要修改代码。（这段还是感觉翻的不太合适）。

**重要提示**:

虽然Core Dta支持SQLite作为一种存储类型，但它不能使用任意的SQLite数据库。Core Data在使用的过程种自己创建这个数据库。（详情，请参考"Persistence Store Features"）。

**持久化文档（Persistent Documents）**

你可以通过代码的方式创建和配置持久存储栈，但在多数情况下，你只是想创建一个基于文档 的应用程序（Document-based application，这个是mac上的）来读写文件。这时，用NSDocument的子类NSPersistentDocument可以让你感受到使 用Core Data的便利。默认状况下，NSPersistentDocument就已经创建了它自己的持久存储栈，其中包含了上下文，和单个的持久对象存储，来处 理这样文档和外部数据“一对一”的映射关系。

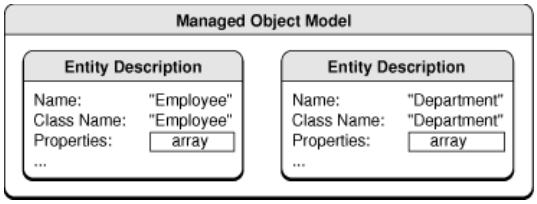
NSPersistentDocument类提供了访问文档的上下文的方法，也实现了标准的NSDocument方法来通过Core Data读写文件。 一般说来，你不需要编写额外的代码来处理对象的持久化。

持久化文档的撤销（undo）操作也被集成在被管理对象的上下文中。

被管理对象和被管理对象模型（Managed Objects and the Managed Object Model）

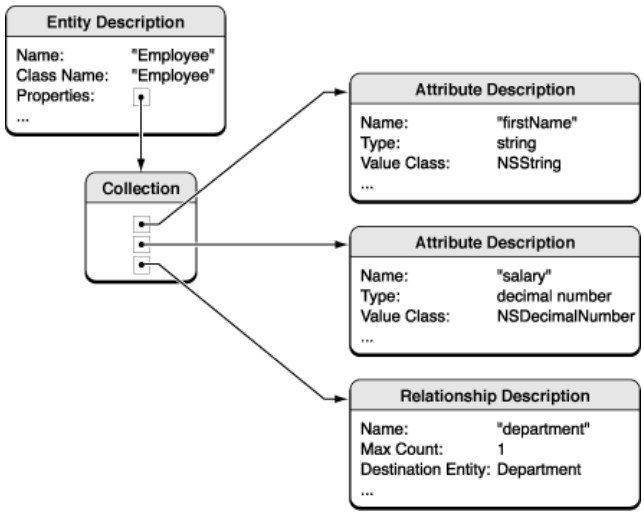
为 了管理对象图，也为了提供对象持久化的功能，Core Data需要对对象有很强的描述能力。被管理对象模型就是程序中对象、实体描述的概要图，如图Figure 5所示。创建模型的常用做法是通过Xcode的图形化建模工具Date Model Design tool。但是如果你愿意的话，也可以在运行时通过代码来建模。

Figure 5 有两个实体的对象模型




模型由多个实体描述对象构成，每个描述提供实体的某项元数据，它们包含实体名、实体在程序中的类名（当然，类名和实体名不需要一致）、属性还有关系。属性和关系依次被属性和关系描述对象所代表，如图Figure 6所示。

Figure 6 带有两个属性和一个关系的的实体描述




被管理对象必须是NSManagedObject或其子类的实例。 NSManagedObject可用来表示任何实体。它使用内部私有的存储机制来维护自身的属性，并执行一个被管理对象所必须的基本操作。一个被管理对象 拥有一份实体描述的引用。在使用时，它通过实体描述来找到自身的元数据，包括实体名和属性、关系的信息。你也可以继承NSManagedObject来执 行额外的操作。

**被管理对象模型（Managed Object Models）**

多数Core Data的功能依赖于你创建的，用来描述程序的实体及其属性、关系的模型图。 模型图由NSManagedObjectModel所表示。一般说来，模型的信息越充实，Core Data能提供的功能就越好。 下文讲解了对象模型的特性，以及如何在程序中创建、使用对象模型。

**被管理对象模型的特性**

被管理对象模型是 NSManagedObjectModel的实例。它描述了你在程序中使用的实体的概要信息。（如果读者不了解entity、property、 attribute和relationship的含义，请先查看"Core Data Basics"和"Cocoa Design Patterns"文档中的"Object Modeling"一节）

**实体（Entities）**

模型包含了NSEntityDescription对象，NSEntityDescription对象指代了模型的实体。关于实体由两个重要特征：名称（name）和类名（name of class）。你应该弄清楚实体、实体的类和作为实体实例的被管理对象之间的区别。

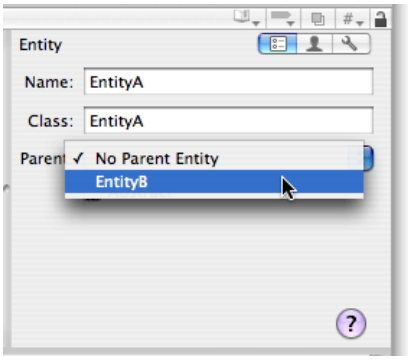
NSEntityDescription 对象可包含NSAttributeDescription对象（指代实体的attribute）和NSRelationshipDescription对 象（指代实体间的relationship）。实体也可能包含fetched属性，该属性由NSFetchedPropertyDescription指 代，模型中有对应的fetch请求的模板，fetch请求由NSFetchRequest所指代。

**实体的继承关系**

实体的继承和类 的继承很类似，当然，也同样有用。 如果你有若干个相似的实体，就可以抽离出它们的共有特性作为一个“父实体”，就省去了在多个实体中都指定相同的属性。 例如，你可以定义一个包含firstName和lastName的“Person”实体，然后在定义子实体"Employee"和"Customer"。

如果是使用Xcode的可视化建模工具来创建模型，你就可以通过如下图的方式为一个实体指定父级实体。

Figure1 Xcode中为一个实体指定父实体




如果你想在代码中创建继承关系。就需要自顶向下来执行。不能直接指定实体的父实体，而只能给一个实体指定子实体（使用setSubentities:）。这 就是说，如果你想给A实体指定父实体，就只能把A作为数组中的一个元素，调用目标父实体setSubentities:的方式来设置。

**抽象实体**

你可以把一个实体指定为“抽象实体”，也就是说，你不打算使用这个实体来创建实例。通常，当你想把这个实体作为父实体，而有子实体来实现详细内容的时候，就 把它声明“抽象实体”。（和抽象类很像）。例如，在一个绘图程序中，你可能会设计一个Graphic实体，它包含了x和y坐标信息、颜色、绘制区域，而你 不

会去创建一个Graphic的实例，而是使用具体的子实体——Circle、TextArea、Line。（这些基本的东西就不给大牛们再罗嗦 了。。。）

**Properties（属性，这个和Attributes的意思一样，实在区别不出来，只好上英语了）**

实体的 Properties是它的attributes和relationship，包含了fetched属性（如果有的话）。每个property都有名称和 类型。 Attribute也可能有默认值。property的名称不能和NSObject和NSManagedObject类中的无参方法名相同。例如，不能把 property命名为"description"。

临时属性（Transient Property）也是作为模型的一部分，但是不作为实体实例的数据保存在持久存储层。Core Data也会跟踪临时属性的变化，以备撤销操作时使用。

注意：如果你用模型外的信息对临时属性执行撤销操作，Core Data将不会使用旧值，调用你的set方法——它只会更新快照信息（snapshot information）。（这段怪怪的，用到的话在修改一下翻译吧）

**Attributes**

Core Data内部支持各种attribute的类型，例如string，date，integer（NSString, NSDate, NSNumber）。如果你使用那些不支持的数据，你需要用到在“Non-Standard Persistent Attributes”介绍到的技术。

你可以将一个attribute声明为“可选”（optional），可选的attribute不 必须有值，但是，不鼓励你将属性置空——尤其是数字值（更好的解决方案是使用强制的值，在这里，我们用默认值，例如0）。 这样做的原因是为了配合SQL中对于空值NULL做比较的操作：NULL不同于Objective-C中的nil。 数据库中的NULL不同于0，搜索0值的操作不会匹配到值为NULL的列。

false == (NULL == 0)

false == (NULL != 0)

而且，在数据库中，NULL也不等于空字符串或是空的数据对象：

false == (NULL == @"")

false == (NULL != @"")

它们之间一点关系都没有。

**关系（Relationships）**

Core Data支持对一、对多的关系，也支持fetched属性。 Fetched property表示了一种“弱”的、单项的关系。 在employees和departments的例子中， department 的一个fetched property可能是“最近雇佣人”（recent hires），而反过来，employee不会拥有这样的关系。

**获取数据请求的模板（Fetch Request Templates）**

我们使用NSFetchRequest类来描述数据请求，利用数据请求从持久存储（persistent store）中获取对象。 经常需要多次执行同样的请求，或是执行某种模式的请求，但是其中包含可变的元素（如查找条件）——这些元素经常有用户提供。 例如，在运行的时候，你要根据用户需要获取某个作者在某个指定日期后的出版的所有出版物。

你可以预定义请求，把它们作为模板存储在被管理对象模型中。 预定义的模板在你需要的时候就可以取出使用。通常情况下，我们通过Xcode的data modeling tool工具创建请求模板。模板可以包含变量，如图Figure 2所示。

Figure 2 Xcode predicate builder


关于Fetch request templates的详细信息，请查看"Accessing and Using a Managed Object Model at Runtime"的描述。

**用户信息字典（User Info Dictionaries）**

模型中的许多元素，诸如entities, attributes, relationships，都有相关的用户信息字典。用熟悉的键－值对，你可以向其中放置任何你需要的数据。这里常用的信息有实体的版本详情，还有针对 fetched property，给谓词（predicate）用的值。

**配置（Configurations）**

配置包含了一个名称和若干个相关的实体。实体的集合是可以重叠的——这就是说，一个实体可以出现在多个配置中。在代码中，我们使用setEntities: forConfiguration:的方法来指定配置。也可以用Xcode的建模工具来指定（选中某个实体，就在属性窗口的第三个，就是一个小扳手的符号）。要获取某项配置的实体，需要用entitiesForConfiguration:的方法。

一般说来，如果你想把不同的实体存放在不同的存储中去，就可能用到配置。一个持久化存储助理（persistent store coordinator）只能有一个被管理对象模型。所以，默认情况下，和助理关联的某个存储必须包含同样的实体。要想绕过这个限制，你可以创建一个包含实体子集的模型，然后为每一个子集创建配置，这样一来，使用这个模型创建助理，当你需要添加存储时，可使用不同的配置指定对应的存储属性。当你创建配置的时候，需要记住，不能创建跨存储的关系。

**使用被管理对象模型**

通常可以使用Xcode的建模工具来创建模型（请参考"Create a managed object with Xcode"）。你也可以全部使用代码来创建（请参考"Core Data Utility Tutorial"）。

**编译数据模型**

数据模型是一种部署资源。 在模型中，除了有实体和属性的详细信息外，用Xcode创建的模型还包含了一些额外的视图信息，包括布局、颜色等等。这些信息在运行时不是必须的。模型文件在编译的过程中会删除这些额外信息以保证尽可能高效的加载。xcdatamodel“源”文件会被momc编译器编译为mom的目标文件。

"mom" 位于 /Library/Application Support/Apple/Developer Tools/Plug-ins/XDCoreDataModel.xdplugin/Contents/Resources/，如果你想把它用在自己的 build脚本中，格式是：mom source destination， source就是Core Data Model文件，destination就是输出的mom文件。

**加载数据模型**

在一些情况下，你不需要写任何加载模型的代码。如果你使用基于文档的程序框架（Document-based application），NSPersistentDocument会管理诸如查找模型、加载模型的任务。 如果你创建了非Document-based application，而且里面又用到了Core Data，一般将获取模型的代码放在application delegate里。模型的存储名称——也就是文件名，

和运行时的名称是不相关的，一旦模型被加载，文件名就没有什么意义了。也就是说，对模型文件，你可以随意命名。

如果你想手动加载模型，有两种方式可用，它们各有各的好处：

你可以从指定的bundle集合里创建整合模型，使用如下的类方法：

mergeModelFromBundles:

也可以用指定的URL加载单个的模型，使用如下的实例方法：

initWithContentsOfURL:（这个方法相信大家都用过）

若不需要考虑分开加载模型，第一个类方法很适用。例如：在你的程序中和程序链接的framework里都有你想要加载的模型。这个类方法可以让你很轻松的加载所有的模型，而不需要考虑模型文件的名称，也不用特定的初始化方法来保证所有的模型都被找到。

但是当你有多个模型要加载，特别是这些模型都代表了一个schema的不同版本，这时，知道要加载哪个模型就很重要了（合并包含相同实体的模型可能导致命名冲突和错误，我们之前“一锅端”的方法不太合适了）。在这种情况下，我们可以用第二个实例方法。 另外，有时我们也需要将模型存储在bundle之外，也需要用这个方法从指定的URL位置加载模型。

还有一点需要说明：我们还有一个类方法 modelByMergingModels:可以用。像mergedModelFromBundles:方法一样，它也能合并给定的若干个模型。这样，我们就可以通过URL来逐一加载模型，然后在创建助理对象之前将它们整合为一个。

**改变模型**

由于模型描述了存储层数据的结构，任何改变模型的动作都将使其不在适配于之前创建的存储层。 如果你改变了模型的结构，就需要将当前存储层的数据迁移到新版本。（请参考"Core Data Model Versioning and Data Migration Programming Guide"文档）。例如：如果你添加了新的实体，或新的属性，你将无法打开旧的存储；如果你添加了验证的限制，或者为属性添加了新的缺省值，你就可以打开旧的存储。

**在运行时访问和适用被管理对象模型**

在运行时，被管理对象模型就是一个简单的“对象图”（这个概念之前提到过），认识到这点很重要，尤其是当你需要用代码来访问模型的详细信息时。例如：修改模型（你只能在runtime之前这样做，请参考 NSManagedObjectModel），取回信息（如本地化实体名，属性数据类型，或数据请求模板）。

在运行时访问模型有很多方法，通过持久栈最终从持久化存储助理得到模型，代码如下：

```
[[aManagedObjectContext persistentStoreCoordinator]managedObjectModel];
```

你也可以通过实体描述得到模型，因此给定一个被管理对象，你就可以得到它的实体描述，进而获得模型。代码如下：

```
[[aManagedObject entity] managedObjectModel];
```

某些情况下，你要维护模型的“直接”引用，也就是说，一个直接返回模型的方法。NSPersistentDocument提供了 managedObjectModel方法，可以返回一个模型，该模型和在文档的上下文中使用的持久化存储助理相关联。如果你使用Core Data Appplication的模板，application delegate将负责模型的引用。

**通过代码创建获取数据请求模板（Fetch Request Templates）**

你可以通过代码创建数据请求模板并将其和模型关联，方法是：setFetchRequestTemplate: forName:如Listing-1所示。 提醒一下：你只能在模型被助理（coordinator）使用之前修改它。

Listing 1 通过代码创建获取数据请求模板

```
NSManagedObjectModel *model = …;
NSFetchRequest * requestTemplate = [[NSFetchRequest alloc]init];
NSEntityDescription *publicationEntity =
   [[model entitiesByName] objectForKey: @"Publication"];
[requestTemplate setEntity: publicationEntity];

NSPredicate *predicateTemplate = [NSPredicate predicateWithFormat:
   @"(mainAuthor.firstName like[cd] $FIRST_NAME) AND \
```

```
        (mainAuthor.lastName like[cd] $LAST_NAME) AND \
        (publicationDate > $DATE)"];
[requestTemplate setPredicate: predicateTemplate];

[model setFetchRequestTemplate: requestTemplate
    forName: @"PublicationForAuthorSinceDate"];
[requestTemplate release];
```

**访问请求模板**

你可以用"Accessing and Using a Managed Object Model at Runtime"里介绍的代码片段来获取并使用请求模板。替换字典必须包含和模板中定义的变量对应的键。如果你想测试null值，必须使用NSNull对象——参考"Using Predicates"。（注：这里的替换字典很好理解，之前的模板中用到了诸如$FIRST_NAME, $LAST_NAME, $DATE这些东西，就相当于我们在模板中创建好的“变量”，我们需要把一个模板“具体化”，就用替换字典，将里面的变量对应一个值，这里看代码就明白了。）

```
NSManagedObjectModel *model = ...;
NSDictionary *substitutionDictionary = [NSDictionary dictionaryWithObjectsAndKeys:
    @"Fiona", @"FIRST_NAME", @"Verde", @"LAST_NAME",
    [NSDate dateWithTimeIntervalSinceNow: -31356000], @"DATE", nil]; //这里的FIRST_NAME, LAST_NAME, DATE和我们之前模板里的$FIRST_NAME, $LAST_NAME和$DATE对应
NSFetchRequest *fetchRequest =
    [model fetchRequestFromTemplateWithName: @"PublicationForAuthorSinceDate"
        substitutionVariables: substitutionDictionary]; //从之前的model中拿出请求模板，然后设定替换字典
NSArray *results =
    [aManagedObjectContext executeFetchRequest: fetchRequest error: &error];
```

要是模板里不包含可替换的变量，你要么

1. 使用fetchRequestFromTemplateWithName: substitutionVariables: 方法，传递nil给第二个参数

或者：

2. 使用fetchRequestTemplateForName: 并将结果copy。这个方法不需要传递“替换变量”这个参数，但是如果你要用返回值本身，将会有异常抛出（无法在不可变的模型中修改命名的数据请求"Can't modify named fetch request in an immutable model"）。

**本地化被管理对象模型**

你可以对模型的大部分内容做本地化处理，包括实体和属性名，还有错误信息。要明白，“转成你自己的语言”也是本地化的一部分。 即使你不打算提供外语版本， 显示“自然语言”的出错提示信息也会有更好的用户体验。例如：“First Name is a required property”就比"firstName is a required property"更好。（后面的这个更像是开发者用的log，显示的是变量名，这里不太明显）。

要想对模型进行本地化处理，需要提供一个本地化字典，模式如下：

Table 1 针对被管理对象模型的本地化字典键值对应关系：

| Key | Value | Note |
|---|---|---|
| "Entity/NonLocalizedEntityName" | "LocalizedEntityName" | |
| "Property/NonLocalizedPropertyName/Entity/EntityName" | "LocalizedPropertyName" | 1 |
| "Property/NonLocalizedPropertyName" | "LocalizedPropertyName" | |
| "ErrorString/NonLocalizedErrorString" | "LocalizedErrorString" | |

备注：(1)在不同实体中的属性，拥有相同的原始名称，但需要不同的本地化名称，适用于该格式。

我们可以通过localizationDictionary方法来访问本地化字典。注意：在Mac OS X 10.4上，这个方法可能返回nil，除了Core Data为了某些特定目的（如报告本地化的错误描述）延迟加载本地化字典。

**字符串文件**

处理模型的本地化最简单的方法就是创建对应的字符串文件——字符串文件名和模型文件名一直，但是后缀名用.strings。（例如，模型文件名为 MyDocument.xcdatamodel，对应的字符串文件名就为MyDocumentModel.strings；如果模型文件已经包含了 Model后缀，你必须再附加一个Model，所以，如果模型文件名为JimsModel.xcdatamodel对应的字符串文件名为 JimsModelModel.strings）。字符串文件格式和标准字符串文件类似（请参考"Localizing String Resources"），但是对应的键值要遵循Table-1中的规则。

一个模型的字符串文件实例：

```
"Entity/Emp" = "Employee";
"Property/firstName" = "First Name";
"Property/lastName" = "Last Name";
"Property/salary" = "Salary";
```

更详细的示例请参考"NSPersistentDocument Core Data Tutorial"。

**代码实现设置本地化字典**

你可以在运行时设定本地化字典，适用NSManagedObjectModel的setLocalizationDictionary:方法即可。你必须创建一个符合Table-1格式的字典，并把它和模型关联。必须保证在模型被使用（获取或创建被管理对象）之前做这些工作，因为再使用后模型就不可编辑了。 Listing 3演示了创建包含本地化字典的被管理对象模型。实体名称叫“Run”，它有两个属性： "date"和"processID"，分别是date和integer类型。process ID的值不能为负。

**Listing 3 通过代码创建被管理对象模型**

```
NSManagedObjectModel *mom = [[NSManagedObjectModel alloc] init];
NSEntityDescription *runEntity = [[NSEntityDescription alloc] init];
[runEntity setName:@"Run"];
[runEntity setManagedObjectClassName:@"Run"];
[mom setEntities:[NSArray arrayWithObject:runEntity]];
[runEntity release];
NSMutableArray *runProperties = [NSMutableArray array];
NSAttributeDescription *dateAttribute = [[NSAttributeDescription alloc] init];
[runProperties addObject:dateAttribute];
[dateAttribute release];
[dateAttribute setName:@"date"];
[dateAttribute setAttributeType:NSDateAttributeType];
[dateAttribute setOptional:NO];
NSAttributeDescription *idAttribute= [[NSAttributeDescription alloc] init];
[runProperties addObject:idAttribute];
[idAttribute release];
[idAttribute setName:@"processID"];
[idAttribute setAttributeType:NSInteger32AttributeType];
[idAttribute setOptional:NO];
[idAttribute setDefaultValue:[NSNumber numberWithInt:0]];
NSPredicate *validationPredicate = [NSPredicate predicateWithFormat:@"SELF >= 0"];
NSString *validationWarning = @"Process ID < 0";
[idAttribute setValidationPredicates:[NSArray arrayWithObject:validationPredicate]
   withValidationWarnings:[NSArray arrayWithObject:validationWarning]];
[runEntity setProperties:runProperties];
NSMutableDictionary *localizationDictionary = [NSMutableDictionary dictionary];
[localizationDictionary setObject:@"Process ID"
   forKey:@"Property/processID/Entity/Run"];
[localizationDictionary setObject:@"Date"
   forKey:@"Property/date/Entity/Run"];
[localizationDictionary setObject:@"Process ID must not be less than 0"
   forKey:@"ErrorString/Process ID < 0"];
[mom setLocalizationDictionary:localizationDictionary];
```

这段代码写的比较多，这里不再解释了。本地化字典的代码在最后。创建一个符合格式的localizationDictionary，然后用model调用即可。

(209)

现在评论

共2条评论

0 喜欢 社区

请输入你的评论

140

昵称（必填） 发 布

按时间排序 | 新浪微博 | 腾讯微博

生莱 (游客)
还是不懂
3月5日 14:33 顶

nuying117 (游客)
写的很好。
2012-12-15 12:02 顶

友言[?]