从零开始学iOS7开发系列3-我的地盘我做主-Cha23

说明:

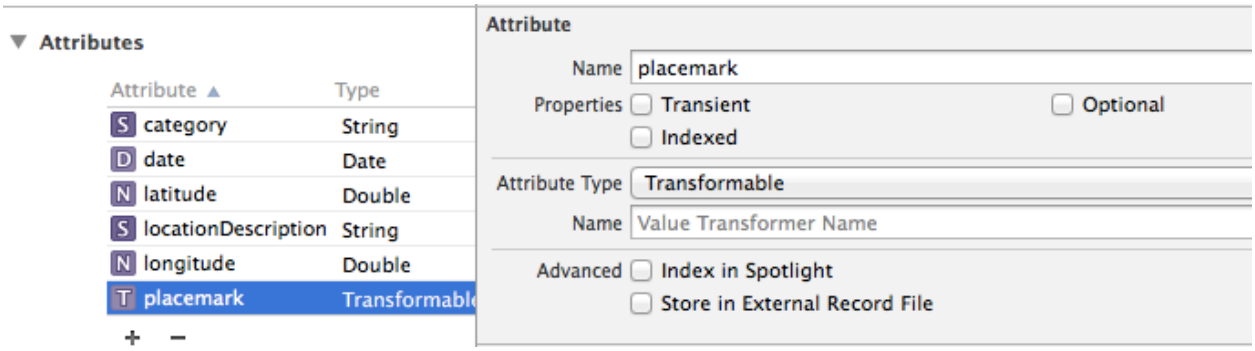
本系列文章的原文及示例代码来自raywenderlich store中的iOS Apprentice 系列3教程，经过翻译和改编。

欢迎继续我们的学习。

谁都不希望我们的应用因为abort()而直接挂掉，至少我们得让用户知道究竟发生了什么。在接下来的内容中，我们会添加一个alert 视图来处理此类情况。当然，这些错误只会在开发过程中发生。但为了避免在真实的用户体验中出现，我们需要采用一种更加优雅的方式来处理~

下面我们采用一种方式来伪造一个致命的错误，以便演示整个过程。

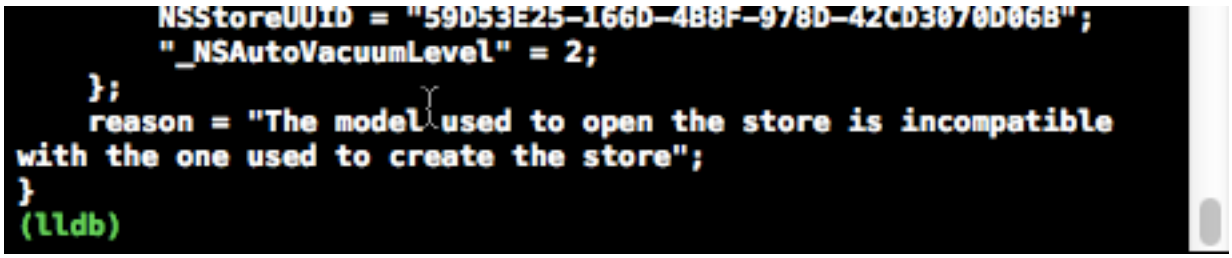
在Xcode中打开数据模型(文件列表中的DataModel.xcdatamodeld），然后选择placemark这个属性，在右侧的inspector选项中取消勾选Optional标记。



当然，这意味着location.placemark永远不会是nil。这是Core Data将要强制约束的一个限制。当我们尝试将一个Location对象保存到数据存储时，因为placemark属性是nil,那么Core Data就会跑出一个错误。这正是我们这里要实现的效果。

编译运行应用，好吧，如你所愿，应用崩溃了~

错误来自AppDelegate中的persisitentStoreCoordinator方法，debug调试区会显示错误的原因如下:



刚才通过对placemark属性进行修改，我们已经调整了数据模型。但这些更改只是针对application bundle（应用束）中的数据模型，而不是Documents文件夹中的数据存储。也就是说DataStore.sqlite文件和更改后的数据模型不再匹配。

有两种方式来修复这个问题：
（1）将DataStore.sqlite文件从Documents文件夹中删掉
（2）从Simulator中删掉整个应用。

好了，限制删除DataStore.sqlite,再次编译运行应用。

当然，这并非是真正要展示给大家的崩溃场景，但非常重要的一点是，一旦我们在开发的过程中更改了数据模型，就有必要删掉之前的数据存储文件，否则Core Data将不能正常工作。

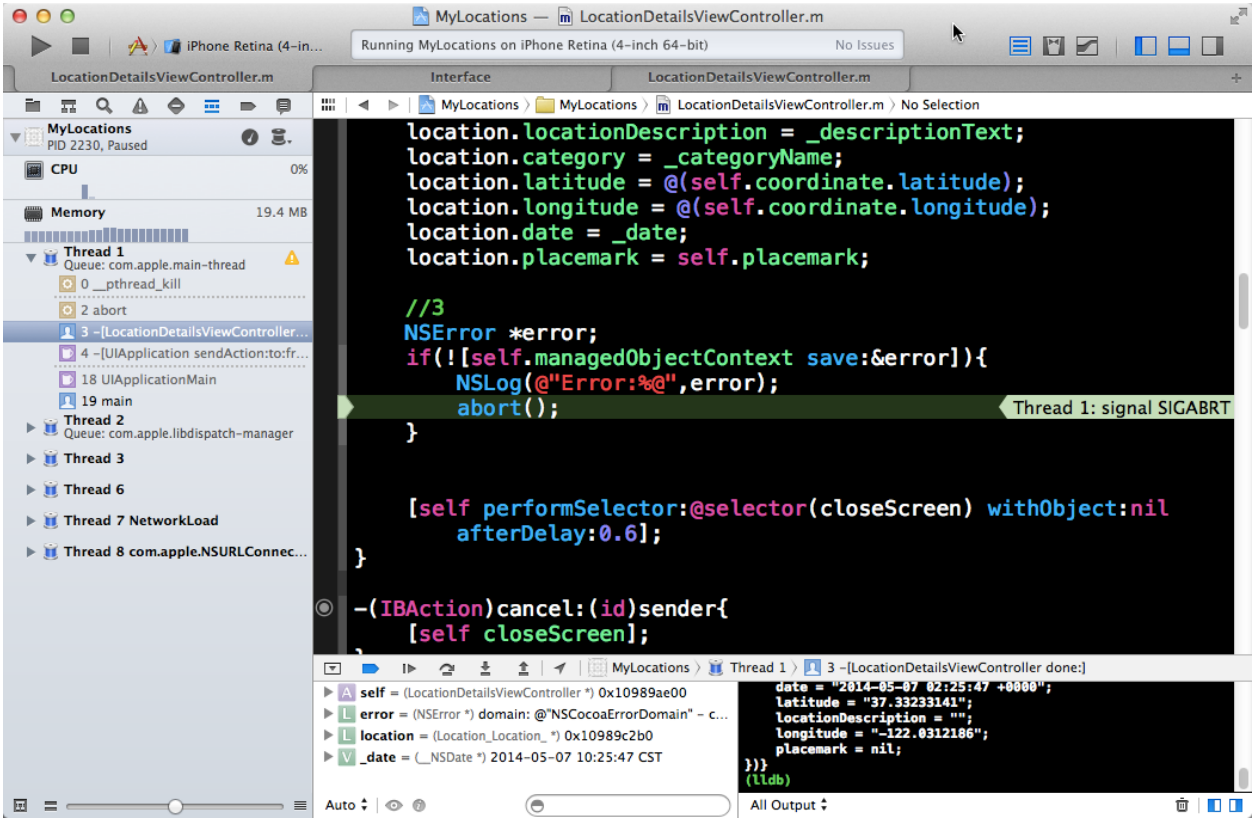
注意：在Core Data中有一个migration机制，当我们使用新的数据模型更新了应用时会非常有帮助。通过使用这种机制，用户的现有数据存储内容会转换成新的格式，而不会导致崩溃。不过在开发的过程中我们就采用最简单的方法，删掉旧的数据库。

这里有一个小小的技巧。触碰Get My Lcoation按钮，然后立即触碰Tag Location。如果你的速度足够快，就可以激活反向地址编码，此时Tag Location界面上会显示：No Address Found。当然，只是在placemark为nil的时候才会显示这样的结果。

如果你觉得geocoding的过程太快，可以暂时注释掉_placemark = [placemarks lastObject];这行代码，它的位置在CurrentLocationViewController.m中的didUpdateLocations方法中。通过这种方式，也会让结果显示无法找到地址。

触碰Done按钮以保存新的Location对象。

此时应用会在调用abort()的时候崩溃：



在debug区域可以看到这样的描述：

NSValidationErrorKey=placemark

这意味着placemark属性并没有得到有效的设置。原因是因为我们在此前已经将其设置成non-optional类型的，因此Core Data不会接收一个nil的placemark值。

对于开发者来说，当应用崩溃时，debugger会接管界面，并指向出错行的代码。可惜用户看到的只是崩溃，崩溃，崩溃。。。~

停止应用。现在尝试从Simulator中触碰应用的图标，从而在Xcode的外部启动应用。重复相同的操作流程让应用崩溃。应用只会停止工作，然后从界面中消失。

想想看，如果一个用户花了0.99美元（6元）甚至更多来买你的这款应用。然后他们看到了这悲催的一幕，你的RP会在此刻败光。即便用户是免费下载了你的应用，他们也一定会劝阻其他朋友下载这个应用。

因此，我们有必要在发生这样的一幕时显示一个alert view视图。当用户取消了alert视图后，我们仍然可以调用abort()来让应用崩溃，但至少用户知道是因为什么原因导致的崩溃。（在alert信息中甚至可以邀请用户联系我们，从而可以更好的解释，以及在下一个版本中修复这一问题）。

在LocationDetailsViewController的done方法中替代原有的错误处理代码：

```
    NSError *error;
    if(![self.managedObjectContext save:&error]){
//        NSLog(@"Error:%@",error);
//        abort();
        FATAL_CORE_DATA_ERROR(error);
        return;
    }
```

这里我们看到，一行FATAL_CORE_DATA_ERROR()的代码取代了之前的NSLog()和abort()。它看起来像一个函数，但实际上是传说中的macro（宏）

在MyLocations-Prefix.pch中添加一个宏定义如下：

```
extern NSString * const ManagedObjectContextSaveDidFailNotification;
#define FATAL_CORE_DATA_ERROR(__error__)\
NSLog(@"*** Fatal error in %s:%d\n%@\n%@",\
__FILE__,__LINE__,error,[error userInfo]);\
[[NSNotificationCenter defaultCenter]postNotificationName:\
ManagedObjectContextSaveDidFailNotification object:error];
```

注意：
在敲入上面的代码时，不应更改#define后面的换行，每行代码后面都需要加上一个反斜杠。

记住一点，我们在Prefix文件中所定义的一切东西都可以在所有的源代码中可见，包括这里的宏。

理论知识充电-宏和预处理

在编译器处理源文件之前，Xcode首先会向它们提供preprocessor（预处理器）。它的作用是处理所有的#import 语句，以及以#标记开头的语句。

当预处理器碰到#import “MyClass.h”这样的语句时，会读取MyClass.h中的所有代码，并插入到源代码中。接下来包含了源代码和所有.h头文件的代码会被移交给编译器。

除了#import语句，预处理器还会处理#define开头的宏定义。在Objective-C中用到宏定义的频率并不高，不过我们还是经常会在各类源代码中看到。

例如，当我们看到下面的语句时：

```
#define SOME_NUMBER 123

- (void)myMethod {

int temp = SOME_NUMBER;

}
```

预处理器会把它转换为：

```
- (void)myMethod {
int temp = 123;
}
```

因此，#define宏定义通常用来为数字常量提供符号化的名称，尽管我们没有看到const这个关键词。
通常情况下，宏定义的名称都是ALL_CAPS_AND_UNDERBARS这样的形式。

宏定义也可以采取很复杂的形式，甚至可以有参数，就像一个函数一样。不过函数和宏定义的区别在于的那个源代码被编译时宏定义会被“扩展”。当编译过程完成后，宏定义就不复存在了。我们不能在程序运行的时候“调用”一个宏定义。宏定义只是为了节省敲代码的时间，以及提升代码的可读性。

通过上面的宏定义，当我们在代码中放入FATAL_CORE_DATA_ERROR()时，预处理器就会将其插入到实际的代码中。

此时的代码就会变成：

```
NSError *error;
if (![self.managedObjectContext save:&error]) {
NSLog(@"*** Fatal error in %s:%d\n%@\n%@",
__FILE__, __LINE__, error, [error userInfo]);
[[NSNotificationCenter defaultCenter] postNotificationName:
ManagedObjectContextSaveDidFailNotification object:error];
}
```

如果我们要在几个不同的地方输入上面的代码，显然是一件很无趣的事情，而通过宏定义可以帮我们节省一点工作，毕竟程序猿就是在偷懒中成长的~

好吧，接下来让我们看看这个宏定义的代码究竟是干什么用的。

首先，它利用NSLog()函数将错误信息输出到Debug 区域：

```
NSLog(@"*** Fatal error in %s:%d\n%@\n%@",
__FILE__, __LINE__, error, [error userInfo]);
```

这部分和我们之前所做的事情类似，除了这里新增加了一个__FILE__和__LINE__。这是由预处理器所提供的两个特殊符号，分别用来代指源文件的名称和行编号。通过使用这两个符号，NSLog()不但可以提供错误，还能够指出错误的所在位置。

在获取了错误信息后，宏定义还将执行以下操作：

```
[[NSNotificationCenter defaultCenter] postNotificationName:
ManagedObjectContextSaveDidFailNotification object:error];
```

我之前曾用”notification”来代指iOS发送的任何通用事件或信息。但在iOS中还有一个名为NSNotificationCenter的对象（别把它和你手机上的Notification Center弄混了~）

以上代码使用NSNotificationCenter来发送一个notification通知。应用中的任一对象都可以订阅此类通知，而当这些事件发生时，NSNotificationCenter会对这些监听对象调用一个特殊的方法。使用这种官方的通知系统是对象彼此之间交流的另一种方式。这种方法的妙处在于，发送通知的对象和接收通知的对象对彼此一无所知。

UIKit中还定义了很多标准的通知让开发者使用。例如，我们可以通过某种通知来了解应用当前是否要被悬停。当然，我们也可以定义自己的通知，也就是这里所做的事情。我们的定制化通知被称为ManagedObjectContextSaveDidFailNotification。

我们采取的办法是，在应用的某个地方监听该通知，然后弹出一个alert view视图，然后调用abort函数。使用NSNotificationCenter的好处是Core Data不需要关心这些东西。每当发生一个保存错误时，不论在应用的哪一点发生，FATAL_CORE_DATA_ERROR()宏会发送该通知，然后某些对象会监听这个通知，并处理该错误。

好吧，那么究竟谁来负责处理这个错误？

app delegate貌似是一个不错的选择。它是应用中级别最高的对象，只要应用还在运行，我们就可以确保该对象的存在。

在AppDelegate.m中的didFinishLaunchingWithOptions方法中添加以下代码（在return语句之前）：

```
    [[NSNotificationCenter defaultCenter]addObserver:self
selector:@selector(fatalCoreDataError) name:ManagedObjectContextSaveDidFailNotification
object:nil];
```

以上代码的作用是，告诉NSNotificationCenter，只要有一个ManagedObjectContextSaveDidFailNotification消息，那么AppDelegate都应该得到通知。selector中包含了要触发的方法的名称。

在AppDelegate.m中添加以下方法：

```
-(void)fatalCoreDataError:(NSNotificationCenter*)notification{

    UIAlertView *alertView = [[UIAlertView alloc]
```

```
        initWithTitle:NSLocalizedString(@"Internal Error", nil) message:NSLocalizedString(@"There was a fatal error in the app and it cannot continue.\n\nPress Ok to terminate the app.Sorry for the inconvenience.", nil)
        delegate:self
        cancelButtonTitle:NSLocalizedString(@"OK", nil)otherButtonTitles:nil, nil];
    [alertView show];


}
```

以上方法的作用就是显示一个alert view视图。这里我们将alert view的delegate设置为self,因此我们需要让AppDelegate遵从UIAlertViewDelegate协议。
在类扩展部分添加该协议：

```
@interface AppDelegate()<UIAlertViewDelegate>
```

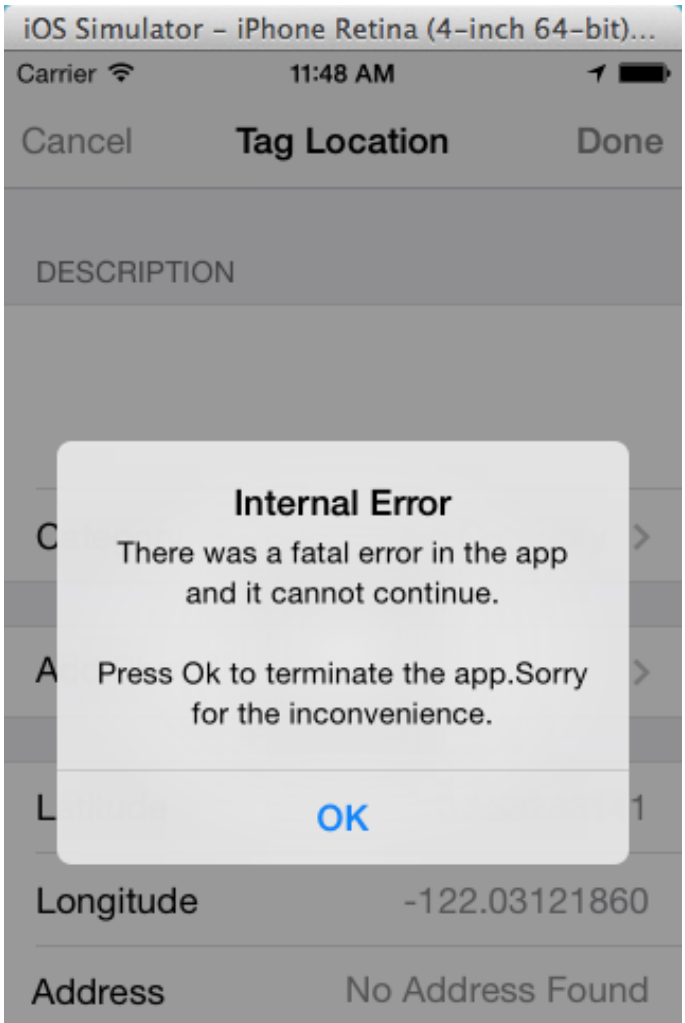
然后实现相应的delegate方法：

```
#pragma mark -UIAlertViewDelegate
-(void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex{
    abort();
}
```

此时我们还有一件事情要做，因为当我们创建了自己notification的时候，还需要在某处给它一个定义。
在AppDelegate.m的顶部添加一行代码：

```
NSString *const ManagedObjectContextSaveDidFailNotification =
@“ManagedObjectContextSaveDidFailNotification";
```

再次编译运行应用，然后尝试在获取街道地址前标记一个location。在应用崩溃前，至少它会告诉用户发生了什么。



当然，我需要再次提醒，我们必须确保没有向Core Data提供任何没有验证的对象。理想的情况是，用户永远不会看到这个alert view，但我们还是得预备这么个东西，毕竟世上没有万全事！

注意：

我们可以使用[managedObjectContext save:]让Core Data验证用户的输入。我们没有必要让应用在一个不成功的save保存后就直接崩溃，除非这种错误不可预料。

除了”optional”标记外，还有其它针对entity的属性的有效性验证。如果我们让用户输入一些数据，那么最好通过save:方法来执行验证。如果验证结果是NO，显然用户所输入的信息存在某种错误，我们需要对此进行解决。

好了，现在回到数据模型，将placemark属性的设置更改回optional。
注意要手动删除DataStore.sqlite文件！
再次编译运行应用确保一切正常。

好了，今天的学习就到处结束了，献上福利~