第八部分　数据持久化

授课教师：杨文川

数据持久化（4学时）

数据持久化就是将内存中的数据模型对象转换为存储模型，然后再将其保存到内存中的过程。

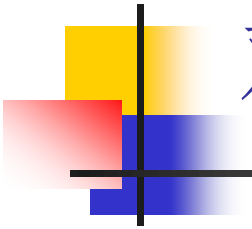
本章目的在于使学生掌握iOS应用的数据持久化技术，包括归档、序列化、SQLite以及Core Data，并通过同一个实例的不同数据持久化实现方法来比较各种技术的适用场景。

- 对象归档
- 属性列表序列化
- Core Data

在前面的实例中，数据都是保存在内存中。一旦退出应用，应用相关数据占用的内存控件就会被释放出来。应用重新启动后，无法读取上次使用时产生的数据，而数据持久化就可以解决这个问题。

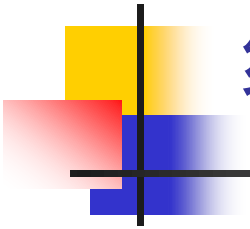
沙箱机制

- 沙箱机制是macOS提供的一种内核级的强制使用的访问控制技术，用来保护系统关键数据和用户数据被盗用和破坏。
- 为了应对类似安全问题，苹果系统设计了沙箱机制。应用的沙箱描述了应用与系统交互的场景。系统授权应用可以访问与其相关的系统资源，从而使应用的任务得以完成。
- 每个应用的沙箱都包含一些固定的目录：Documents、Library以及tmp等。
  - Documents目录用来保存应用运行过程时产生的数据。当设备与iTunes或者iCloud同步时，该目录下的数据会进行备份。
  - Library目录下有Caches和Preferences子目录
  - Caches子目录用来存放应用运行过程中的缓存文件
  - Preferences子目录则用于存放应用程序的设置相关的数据。
  - tmp目录存放应用运行时的临时文件。



对象归档

- 通过归档，可以将复杂的对象及其关系存储下来。归档保留了每一个对象的唯一标识以及与其它对象间的关系。
- 归档的逆过程为解压（unarchive）,归档了的对象通过解压可以提取对象本身及其与其它对象间的关系。
- 归档非常适合用来保存应用的数据模型。Cocoa提供了归档的基础框架，开发者可直接使用，从而大大简化了归档的流程。
  - 一个对象要能进行归档，就必须遵守NSCoding协议。
  - NSCoding协议含有两个重要的方法：
    - 负责将对象中信息编码为比特流数据；
    - 负责从比特流中解码出对象的相关信息。



编码器对象

- 编码器对象负责对象与归档文件之间的读写工作。
- 编码器对象是继承抽象类NSCoder的子类的实例。
- NSCoder声明了一个可扩展的接口：
  - 实现了从对象中获取信息，并将其写入到文件中的过程（编码）；
  - 实现了从比特流中获取对象信息，并将其恢复成一个对象的相反过程（解码）。
- 编码器为了按照特定的格式进行归档，需要实现NSCoder中相应的方法。
- 编码器对象通过向对象发送编码或解码的消息来实现读或写对象的操作。
  - 创建归档时，编码器会发送消息encodeWithCoder:给对象；
  - 读取归档时，编码器会发送消息initWithCoder。
  - 在NSCoding协议中定义了这些消息。要归档一个对象，对象所属的类必须先遵守NSCoding协议。

键值归档器对象

- 键值归档器对象*NSKeyedArchiver：*负责根据对象来创建归档文件
- 键值解档器对象*NSKeyedUnarchiver：*负责将归档文件解码为原来的对象。
- 键值归档文件为每一个变量的值都分配一个唯一的键值。当解码键值归档文件时，可以根据键值来获取相应变量的值。
- 创建归档文件：在归档类中调用类方法archiveRootObject:toFile:或者archivedDataWithRootObject:
- 提取归档文件：在归档类中调用方法unarchiveObjectWithFile:或者unarchiveObjectWithData:

实例：学生成绩表6.0

- 要求：
  - 在前一实例“学生成绩表5.0”的基础上，通过对象归档的方式将用户编辑后的学生成绩表保存下来，并在应用重新启动后将归档文件加载到学生成绩表中。

代码参见iosPrj的Chapter07-StudentPerformance6.0

增加对象归档的功能

- 本例中的学生实例数据需要保存下来，
  - 学生类必须遵守NSCoding协议。
  - 实现NSCoding协议中的两个方法encode(with:)和init(coder:)。

```
import UIKit

class Student: NSObject,NSCoding {

    var name: String
    var score: Int
    var id: String

    init(name: String, score: Int, id: String) {

        self.name = name
        self.score = score
        self.id = id
        super.init()

    }

    func encode(with aCoder: NSCoder) {

    }

    required init?(coder aDecoder: NSCoder) {

    }

}
```

代码参见iosPrj的Chapter07-StudentPerformance6.0

实现方法encode(with:)

- 实现NSCoding协议中的方法encode(with:)
  - 当student类收到消息encode(with:)后，它负责将其所有的属性编码为NSCoder对象。
  - 在保存对象的时候，就可以使用这个NSCoder对象写入到比特流数据中。
  - 而这个比特流数据将会按照“键值对”的方式组织，并保存到文件系统中。

```
func encode(with aCoder: NSCoder) {

    aCoder.encode(name, forKey: "name")
    aCoder.encode(score, forKey: "score")
    aCoder.encode(id, forKey: "id")

}
```

实现方法init(coder:)

- 实现NSCoding协议中的方法init(coder:)：
  - 当student类收到消息init(coder:)后，它负责从归档文件中载入所有对象，并根据键值来给每个属性赋值。

```
required init?(coder aDecoder: NSCoder) {

    name = aDecoder.decodeObject(forKey: "name") as! String

    score = aDecoder.decodeInteger(forKey: "score")

    id = aDecoder.decodeObject(forKey: "id") as! String

    super.init()
}
```

代码参见iosPrj的Chapter07-StudentPerformance6.0

构建归档保存路径

- 完成以上工作后，Student类的实例就可以通过归档的方式被保存或载入到文件系统中了，或者说Student类实例支持被归档了。
- 要实现Student实例的归档，还需要做两件事：
  - 提供归档文件的保存路径；
  - 在StudentsInfo类的初始化方法中载入归档文件并解码到对象实例中，在保存方法中将对象实例编码并写入归档文件
- 由沙箱可知，Student类实例的归档文件将会被保存在“Documents”目录下。
  - 因此，要提供一个该目录下的保存路径，即：URL。

```
let archiveURL: URL = {

    let documentsDirectories =
        FileManager.default.urls(for: .documentDirectory,
                                 in: .userDomainMask)

    let documentDirectory = documentsDirectories.first!

    return
        documentDirectory.appendingPathComponent("students.archive")

}()
```

保存对象实例

- 构建了保存路径后，就可以保存和载入对象实例了。由前面的介绍可知，保存对象实例是通过NSKeyedArchiver来实现的，而载入对象实例是通过NSKeyedUnarchiver来实现的。
- 在StudentsInfo类中增加一个方法saveStudents()，用来将所有的students实例对象保存到归档文件中，

```
func saveStudents() -> Bool {

    return NSKeyedArchiver.archiveRootObject
        (studentsCollection, toFile: archiveURL.path)
}
```

触发归档动作

- 在应用运行的过程中是不应该保存对象实例的，因为在运行过程中随时可能修改对象实例的数据，这将导致频繁的触发归档动作，而归档是需要占用系统资源的。
- 触发归档动作的时机应该是用户按HOME键的时候，此时应用将进入后台运行状态，应用的数据随时可能因为内存空间紧张而被释放。
- 应用在进入后台运行状态时，应用会收到消息applicationDidEnterBackground(_:)，应在此处调用归档方法saveStudents()。
- 在方法applicationDidEnterBackground(_:)中调用保存归档文件的方法saveStudents()。

AppDelegate.swift

```
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    let studentsInfo = StudentsInfo()


    func application(_ application: UIApplication,
        didFinishLaunchingWithOptions launchOptions:
        [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // Override point for customization after application launch.

        //let studentsInfo = StudentsInfo()

        let navigationController = window!.rootViewController as!
            UINavigationController

        let studentPerformanceTableViewController = navigationController.
            topViewController as! StudentPerformanceTableViewController

        studentPerformanceTableViewController.studentsInfo = studentsInfo

        return true
    }

func applicationDidEnterBackground(_ application: UIApplication) {

    if (studentsInfo.saveStudents()) {
        print("students are saved.")
    } else {
        print("Fail to save students!")
    }

}
```

载入归档文件

- 修改类StudentsInfo中的初始化方法：
  - 将原来的初始化代码删除。
  - 通过键值解档器对象NSKeyedUnarchiver的方法.unarchiveObject从指定的路径中取得归档文件，并将其解码为Student类型的数组。
- 每次应用启动的时候都会从系统目录中载入归档的文件，并将其解码后赋值给相应的对象，从而完成对象的初始化工作。

```
init() {

    if let theStudents = NSKeyedUnarchiver.unarchiveObject(withFile:
        archiveURL.path) as? [Student] {

        studentsCollection = theStudents

    }
}
```

代码参见iosPrj的Chapter07-StudentPerformance6.0

运行效果

- 程序启动后，学生成绩表中一共有8条记录。
- 对其进行编辑，删除后6条记录，并加入一条新的记录。编辑后的界面如右图所示。
- 退出应用，并重启系统。再次进入应用后，界面和右图一样，说明编辑后的学生成绩表信息得到了正确的归档和解档。

第八部分　数据持久化

本章目的在于使学生掌握iOS应用的数据持久化技术，包括归档、序列化、SQLite以及Core Data，并通过同一个实例的不同数据持久化实现方法来比较各种技术的适用场景。

- 对象归档
- 属性列表序列化
- Core Data

属性列表序列化

- 序列化是将分层数值对象进行存储的方法。序列化方法保存对象的值以及它们在层次结构中的位置。
  - 分层数值对象： 字典类型、数组类型、字符串型以及二进制数据
- 属性列表就是一种的序列化实例。
  - 应用的设置以及用户偏好都采用属性列表来保存。
- 序列化方法将对象转化成比特流，而反序列化则将比特流转化为对象。
- 序列化与归档不同之处在于：
  - 序列化不保存数值的类型以及它们之间关系，它只存储数值本身。因此，在进行反序列化的时候，开发者需要来正确的选择数值的类型以及它们的关系。
- 属性列表在序列化时，不保留对象的类标识，只将对象归入字典类型或者数组类型等支持的类型。因此，一个属性列表在进行序列化或反序列化时，目标属性列表中的对象类型可能会发生变化。
- 属性列表的序列化也不会保存原来对象中的引用关系。归档方式更适合于对象的持久化，而属性列表更适合保存结构化的数据。

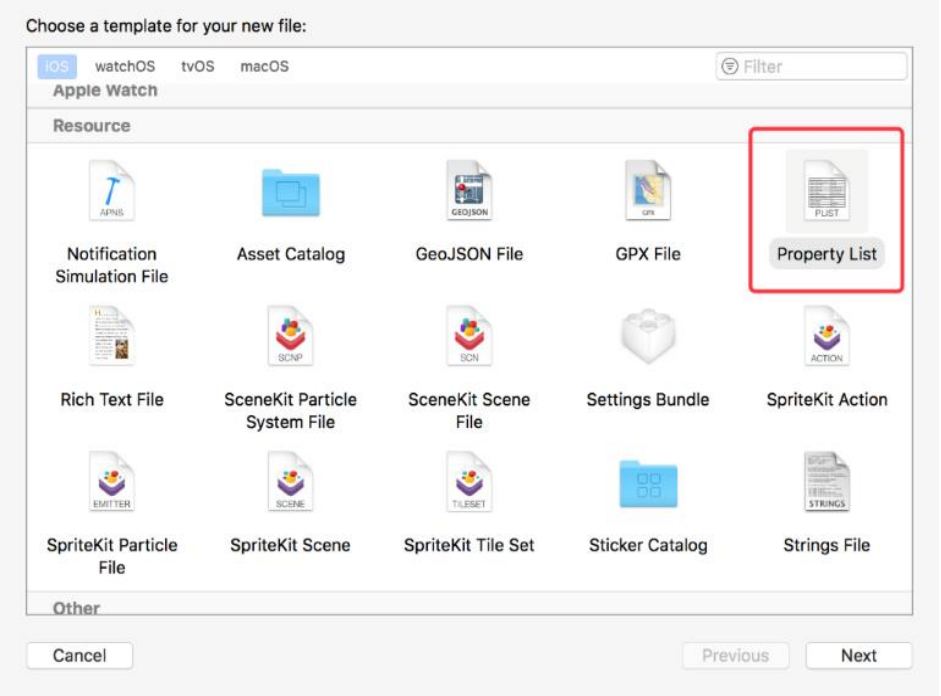
实例：学生成绩表6.1

- 要求
  - 在前例“学生成绩表5.0”的基础上，通过属性列表的方式将用户编辑后的学生成绩表保存下来，并在应用重新启动后将属性列表加载到学生成绩表中。

代码参见iosPrj的Chapter07-StudentPerformance6.1

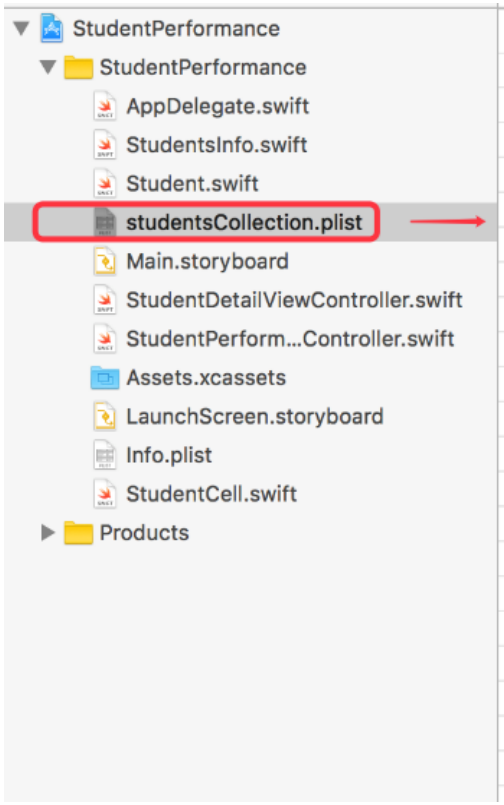
创建属性列表文件

- 创建一个缺省的属性列表文件studentsCollection.plist，用来保存学生成绩表的初始化数据。
  - 在文件模板选择窗口中找到“Resource”标签，选择其中的“Property List”，创建一个新的属性列表文件studentsCollection.plist。



将数据写入属性列表文件

- 创建完属性列表文件后，将缺省的学生成绩数据信息加入到其中



Key	Type	Value
Root	Array	(5 items)
Item 0	Dictionary	(3 items)
name	String	Tommy
score	String	98
id	String	37060101
Item 1	Dictionary	(3 items)
name	String	Jerry
score	String	65
id	String	37060102
Item 2	Dictionary	(3 items)
name	String	Kate
score	String	78
id	String	37060103
Item 3	Dictionary	(3 items)
name	String	Jack
score	String	85
id	String	37060104
Item 4	Dictionary	(3 items)
name	String	Ben
score	String	100
id	String	37060105

读取属性列表文件

- 打开项目文件StudentsInfo.swift，重新编写初始化函数init()
- 在初始化函数中，通过属性列表来对studentsCollection进行初始化。
- 每次程序启动时，StudentsInfo类都会进行初始化。
  - 如果程序是第一次启动，则需要将项目资源中的plist文件复制到沙箱目录下；
  - 如果程序不是第一次启动，那么直接将沙箱目录中的plist文件取出即可。
- init()中先判断沙箱目录中是否已经存在plist文件。
- 然后读取沙箱目录中的plist文件，并将其赋值给一个数组theCollection。
- 接着将数组中的每一个元素转化为字典类型。
- 然后再根据键值对取出theStudent对象的属性值
- 最后将其作为元素添加到studentsCollection数组中。

```
init() {

    let fileExist = FileManager.default.fileExists(atPath: plistURL.path)

    if (!fileExist) {

        let bundlePath = Bundle(for: StudentsInfo.self).resourcePath as
            NSString?

        let thePath =
            bundlePath!.appendingPathComponent("studentsCollection.plist")

        do {

            try  FileManager.default.copyItem(atPath: thePath, toPath:
                plistURL.path)

        } catch {

            print("fail to copy plist file!")
        }
    }

    let theCollection = NSMutableArray(contentsOfFile: plistURL.path)!

    for theElement in theCollection {

        let dict = theElement as! NSDictionary

        let name = dict["name"] as! String

        let score = dict["score"] as! String

        let id = dict["id"] as! String

        let  theStudent = Student(name: name, score: Int(score)!, id: id)

        studentsCollection.append(theStudent)
    }

}
```

构建属性列表文件的存储路径

```
let plistURL: URL = {

    let documentsDirectories =
        FileManager.default.urls(for: .documentDirectory,
                                 in: .userDomainMask)

    let documentDirectory = documentsDirectories.first!

    return
        documentDirectory.appendingPathComponent("studentsCollection.plist")

}()
```

代码参见iosPrj的Chapter07-StudentPerformance6.1

设置studentsInfo为全局常量

- 在文件AppDelegate.swift中编辑

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?

    let studentsInfo = StudentsInfo()


func application(_ application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // Override point for customization after application launch.

    //let studentsInfo = StudentsInfo()

    let navigationController = window!.rootViewController as!
        UINavigationController

    let studentPerformanceTableViewController = navigationController.
        topViewController as! StudentPerformanceTableViewController

    studentPerformanceTableViewController.studentsInfo = studentsInfo

    return true
}
```

- 在应用进入后台时保存应用中的相关数据

```
func applicationDidEnterBackground(_ application: UIApplication) {

    if (studentsInfo.saveStudents()) {
        print("students are saved.")
    } else {
        print("Fail to save students!")
    }
}
```

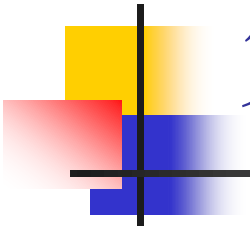
保存数据

- 在文件StudentsInfo.swift中，重新编写方法saveStudents()

```
func saveStudents() -> Bool {

    let studentsArray = NSMutableArray()

    for theStudent in studentsCollection {

        let studentDictionary : NSDictionary

        studentDictionary = ["name":theStudent.name, "id":theStudent.id,
            "score": "\(theStudent.score)"]


        studentsArray.add(studentDictionary)

    }

    studentsArray.write(toFile: plistURL.path, atomically: true)

    return true
}
```

运行效果

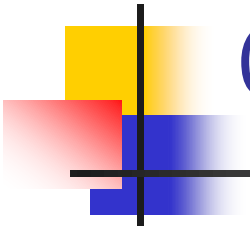

iPhone 7 Plus – iOS 10.3 (14E8301)
Carrier
5:33 PM
Edit
学生成绩表
Tommy 37060101 98
Jerry 37060102 65
Kate 37060103 78
Jack 37060104 85
Ben 37060105 100
iPhone 7 Plus – iOS 10.3 (14E8301)
Carrier
5:32 PM
Edit
学生成绩表
Tommy 37060101 98
Jerry 37060102 65
Kate 37060103 78
Jack 37060104 85
Benny 37060105 1008



第八部分　数据持久化

本章目的在于使学生掌握iOS应用的数据持久化技术，包括归档、序列化、SQLite以及Core Data，并通过同一个实例的不同数据持久化实现方法来比较各种技术的适用场景。

- 对象归档
- 属性列表序列化
- Core Data



Core Data

- 对象归档和属性列表序列化的缺陷
  - 在保存数据时都是对整个文件进行读写。对于频繁更新的数据来说，这两种方法会占用大量的系统资源，并且效率低下。未解决这个问题，需要使用Core Data。
- Core Data可以每次读取对象的一个子集。当对象的某些属性值发生了变化后，可以通过Core Data进行增量更新，这种方式将极大提高应用的性能，并占用极少的系统资源。
- Core Data是一个用来管理应用中模型层中对象的框架。
  - 它提供了一套能够满足大部分需求的对象全生命周期管理解决方案。它不仅保存了数据模型对象，而且还保存了对象间的关系。
  - Core Data通过对象图（Object Graph）来实现对象数据和关系的存储。Core Data的底层数据存储技术是SQLite数据库。
- 对于开发者来说，只需要直接操作Core Data即可，不需要关心底层数据库的细节。使用Core Data开发模型层可以节省50-70%的代码量。

两个重要概念

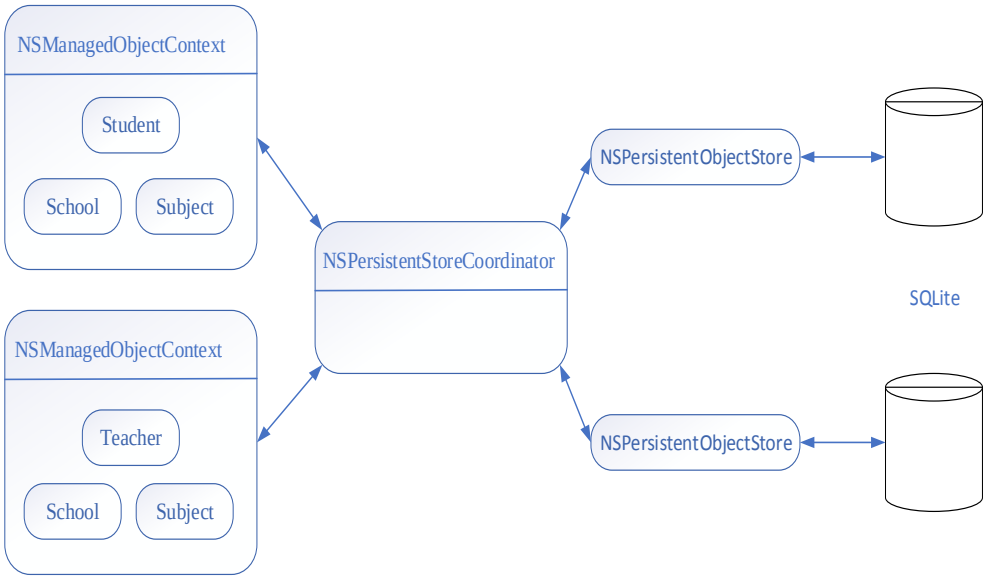
- **NSManagedObject**
- Core Data框架通过托管对象实例NSManagedObject来管理应用中的实体、实体的属性以及实体间的关系。
- NSManagedObject实例与数据模型中的实体之间存在映射关系。
- 实体由实体名、实体的属性以及对应的NSManagedObject类型的类名组成。
- Xcode提供了图形化的实体编辑器，并可以自动创建实体对应的托管对象NSManagedObject。

- **Core Data栈**
- Core Data栈是在初始化时创建的一个数据模型层的NSManagedObject对象集。
- 它负责协调应用中托管对象和外部存储数据的一致性，并处理所有与外部存储数据的交互。
- 使用Core Data作为数据持久化方案，能够使开发者专注于业务逻辑，而不用关心底层数据存储的实现细节。

Core Data栈

- Core Data栈由四个主要的对象组成：
  - 托管对象上下文 NSManagedObjectContext
  - 持久存储协调器 NSPersistentStoreCoordinator
  - 托管对象模型NSManagedModel
  - 持久化容器NSPersistentContainer。
- Core Data栈处于应用程序对象和外部数据存储之间，由持久化存储和持久化存储协调器对象组成。
- 持久化对象存储位于栈的底部，负责外部存储（比如SQLite数据库）数据和托管对象上下文之间的映射关系，但不直接和托管对象上下文交互。
- 在栈的持久化对象存储上面是持久化存储协调器，它为一个或多个托管对象上下文提供访问接口，使其下层的多个持久化存储可以表现为单一存储。

Core Data架构中各种对象之间的关系



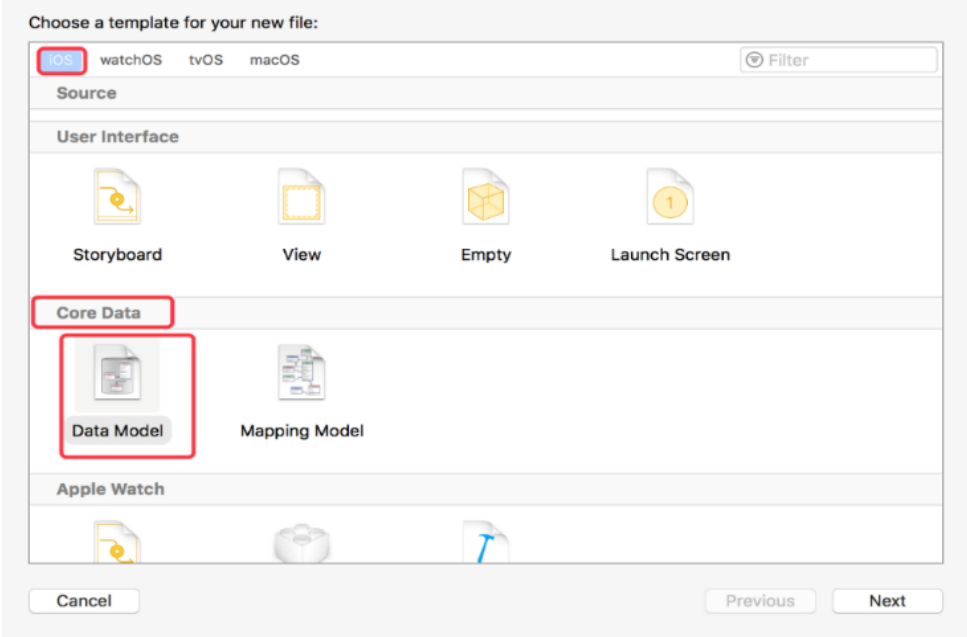
实例：学生成绩表6.2

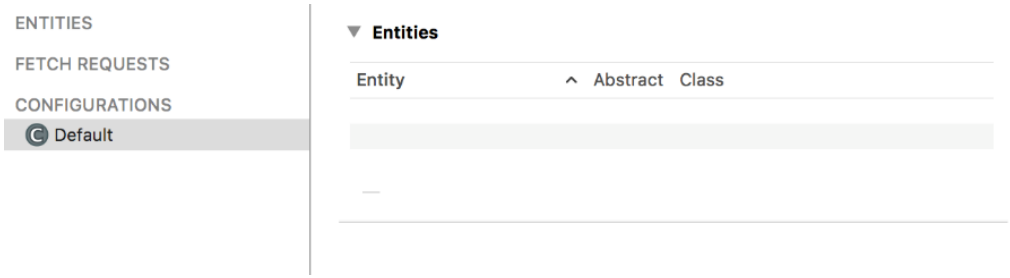
- 要求
  - 在前例“学生成绩表6.1”的基础上，通过Core Data将用户编辑后的学生成绩表保存下来，并在应用重新启动后将Core Data中的数据加载到学生成绩表中。

代码参见iosPrj的Chapter07-StudentPerformance6.2

增加Core Data支持

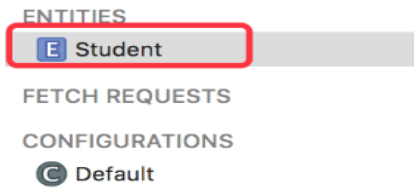
- 删除原有项目中文件Student.swift和studentsCollection.plist。
- 增加Core Data保存数据的功能
  - 手动创建数据模型文件
  - 创建完以后，在项目文件列表的窗口中选择.xcdatamodeld，打开该文件的编辑界面

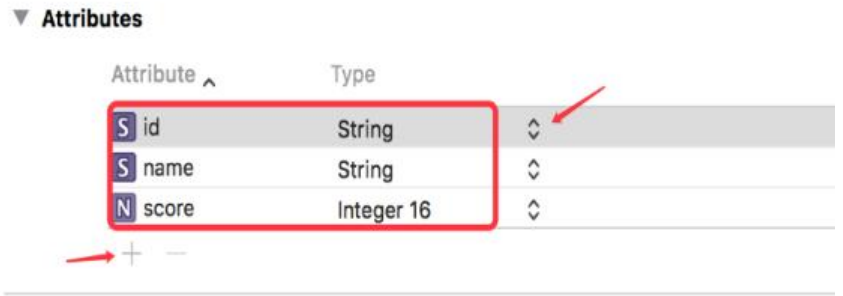


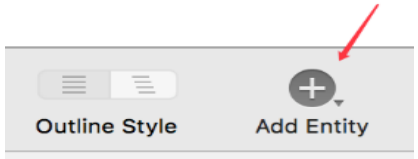


配置数据模型

- 添加实体
- 选中实体，添加属性

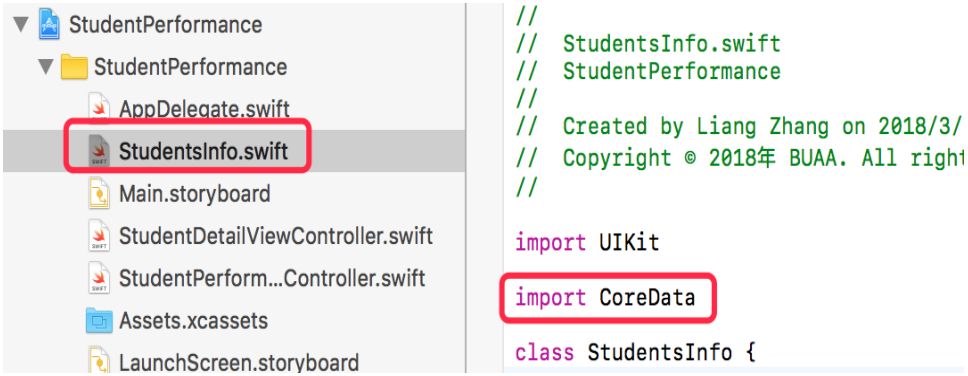






修改数据模型

- 在文件StudentsInfo.swift中加载CoreData包的代码
- 将变量studentsCollection的类型修改为NSManagedObject数组类型，用来接收从CoreData中读取的数据。




```
import UIKit
import CoreData
class StudentsInfo {
    var studentsCollection: [NSManagedObject] = []
```

Core Data数据读取

- 修改其中的tableView(:, cellForRowAt)方法。
- 修改三个标签的赋值方式，这里的student为NSManagedObject类型，需要根据键来取值，并且要指定类型。

```
override func tableView(_ tableView: UITableView, cellForRowAt indexPath:
    IndexPath) -> UITableViewCell {

    let cell = tableView.dequeueReusableCell(withIdentifier:
        "UITableViewCell", for: indexPath) as! StudentCell

    let student = studentsInfo.studentsCollection[indexPath.row]

    cell.nameLabel.text = student.value(forKey: "name") as? String
    cell.idLabel.text = student.value(forKey: "id") as? String
    cell.scoreLabel.text = "\(student.value(forKey: "score") as! Int16)"

    print(student.value(forKey: "score") as! Int16)

    return cell
}
```

添加学生记录

- 修改原来的addStudent(_ sender: UIBarButtonItem)方法。
- 这里修改为：通过弹出警告框来要求用户输入学生的相关信息。其中，生成一个警告框控制器实例alert，并向其中添加三个文本编辑框控件，用来接收用户输入的姓名、学号及成绩信息。
- 定制UIAlertAction实例saveAction，实现将新增学生信息保存到Core Data中。这是通过调用studentsInfo中的addStudent方法来实现的。

StudentsPerformanceTableViewControllerInfo.swift

```
@IBAction func addStudent(_ sender: UIBarButtonItem) {

    let alert = UIAlertController(title: "添加一条学生记录",
                                  message: "请依次输入学生的姓名，学号，成绩",
                                  preferredStyle: .alert)

    alert.addTextField{(textField: UITextField) -> Void in textField.
        placeholder = "姓名"}
    alert.addTextField{(textField: UITextField) -> Void in textField.
        placeholder = "学号"}
    alert.addTextField{(textField: UITextField) -> Void in textField.
        placeholder = "成绩"}

    let saveAction = UIAlertAction(title: "Save", style: .default) { [unowned
        self] action in

        let theName = alert.textFields?[0].text
        let theId = alert.textFields?[1].text
        let theScore = alert.textFields?[2].text

        let theStudent = self.studentsInfo.addStudent(name: theName!, id:
            theId!, score: Int(theScore!)!)

        if let theIndex = self.studentsInfo.studentsCollection.index(of:
            theStudent) {

            let theIndexPath = IndexPath(row: theIndex, section: 0)

            self.tableView.insertRows(at: [theIndexPath], with: .automatic)

        }
    }

    let cancelAction = UIAlertAction(title: "Cancel",
                                     style: .default)

    alert.addAction(saveAction)
    alert.addAction(cancelAction)
    present(alert, animated: true)
}
```

Core Data添加记录

- 向文件StudentsInfo.swift中添加方法addStudent，用来实现添加一条学生记录时，在Core Data数据模型中需要做的工作。

```
func addStudent(name:String, id:String, score:Int) -> Student {

    let managedContext = persistentContainer.viewContext

    let entity = NSEntityDescription.entity(forEntityName: "Student", in:
        managedContext)!

    let theStudent = NSManagedObject(entity: entity, insertInto: managedContext)

    theStudent.setValue(name, forKey: "name")
    theStudent.setValue(id, forKey: "id")
    theStudent.setValue(score, forKey: "score")

    studentsCollection.append(theStudent)

    return theStudent as! Student

}
```

Core Data初始化

- 修改类StudentsInfo的初始化方法：
  - 获取上下文实例和实体Student中的所有记录，并将其赋值给studentsCollection。
- 在类StudentsInfo的声明部分定义一个私有常量persistentContainer，并创建一个数据模型StudentPerformance的持久化容器实例，将其赋值给persistentContainer。

```
init() {

    let managedContext = persistentContainer.viewContext

    let fetchRequest = NSFetchRequest<NSManagedObject>(entityName: "Student")

    do {

        studentsCollection = try managedContext.fetch(fetchRequest)

    } catch let error as NSError {

        print("Could not fetch. \(error), \(error.userInfo)")

    }

}


private let persistentContainer: NSPersistentContainer = {
    let container = NSPersistentContainer(name: "StudentPerformance")
    container.loadPersistentStores(completionHandler: { (storeDescription, error)
        in
        if let error = error as NSError? {
            fatalError("Unresolved error \(error), \(error.userInfo)")
        }
    })
    return container
}()
```

Core Data的CRUD操作

- 添加记录的方法 saveStudents
- 删除记录的方法 deleteStudent

```
func saveStudents() -> Bool {

    let managedContext = persistentContainer.viewContext

    if managedContext.hasChanges{
        do {
            try managedContext.save()
        } catch  {
            let nsError = error as NSError
            print("Error in saving data!", nsError.localizedDescription)
        }
    }

    return true
}


 func deleteStudent(_ theStudent: Student) {

     if let theIndex = studentsCollection.index(of: theStudent) {

         studentsCollection.remove(at: theIndex)

         let managedContext = persistentContainer.viewContext

         managedContext.delete(theStudent)

     }

 }
```

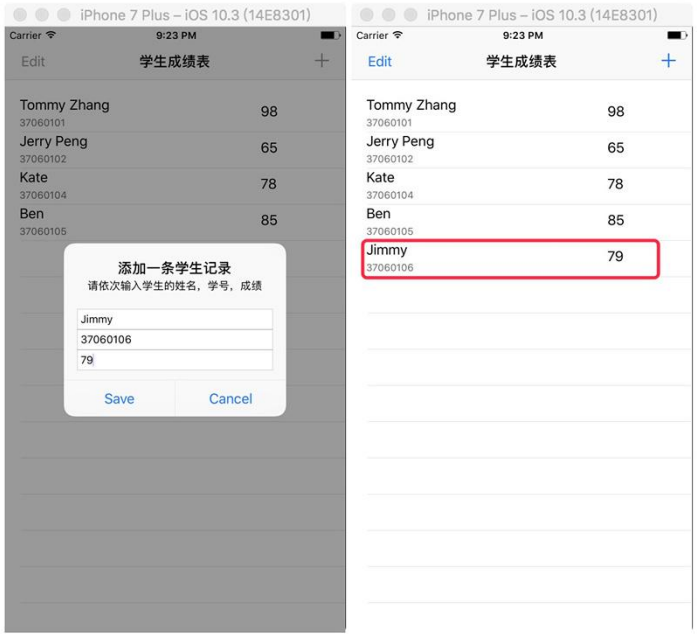
保存结果到Core Data中

- 在文件AppDelegate.swift中，向方法applicationDidEnterBackground中添加代码，调用studentsInfo中的保存上下文的方法saveStudents
- 该方法将在应用进入后台之前被触发，此时统一将持久化容器上下文实例保存到Core Data中。

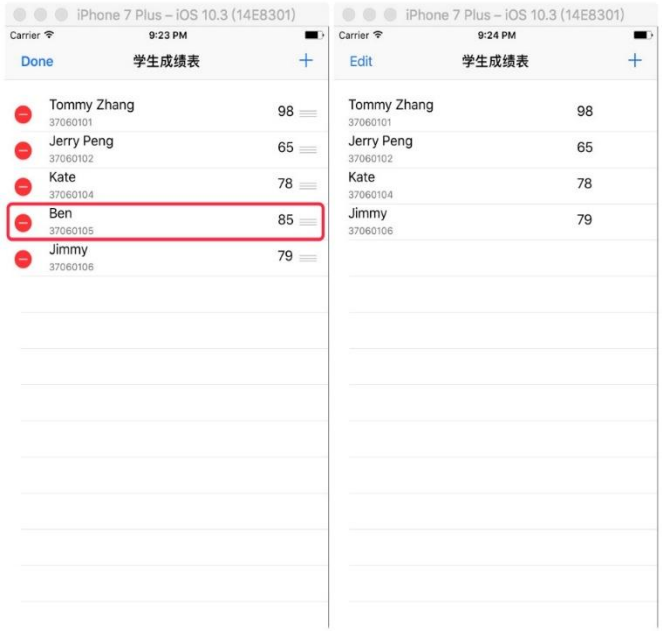
```
func applicationDidEnterBackground(_ application: UIApplication) {

    if (studentsInfo.saveStudents()) {
        print("students are saved.")
    } else {
        print("Fail to save students!")
    }
}
```

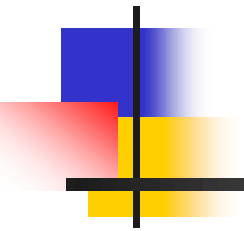
运行效果

打开一个空的表格视图，向其中添加5条学生成绩记录，然后按Home键，切换到主界面，并重启系统，发现添加的记录全部保存下来了，并显示在表格中了



在应用中切换到编辑模式，并将最后一条记录删除。然后按Home键，切换到主界面，并重启系统，发现在表格视图中只剩下了4条记录，被删除的记录不见了





谢　　谢