ByteDance 字节跳动

20秋

MOSAD

现代操作系统应用开发

#09 本地数据存储

Content

- 文件存储
  - 沙盒目录文件存取（NSFileManager）
  - App资源读取（NSBundle）
  - 键值对信息存取（NSUserDefaults）
- 数据库存储
  - SQLite3
  - FMDB
  - CoreData
- 数据安全
  - Keychain
- 对象序列化
  - NSSecureCoding
  - YYModel

文件存储

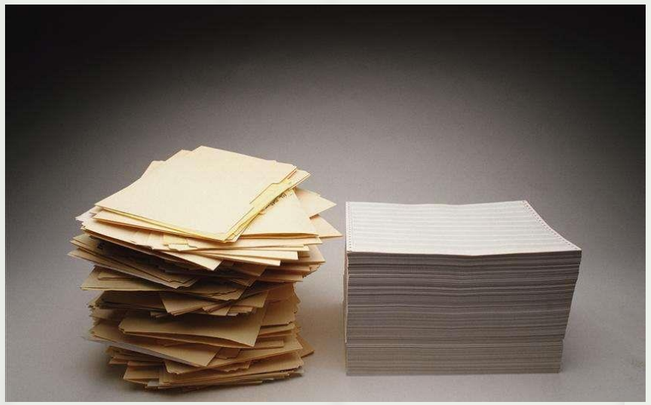
数据存储

- 对象持久存储在文件、沙盒中
- 序列化：将对象转换成二进制数据
- 二进制数据的保存和读取
- 反序列化：将二进制数据转换成对象

对象

序列化

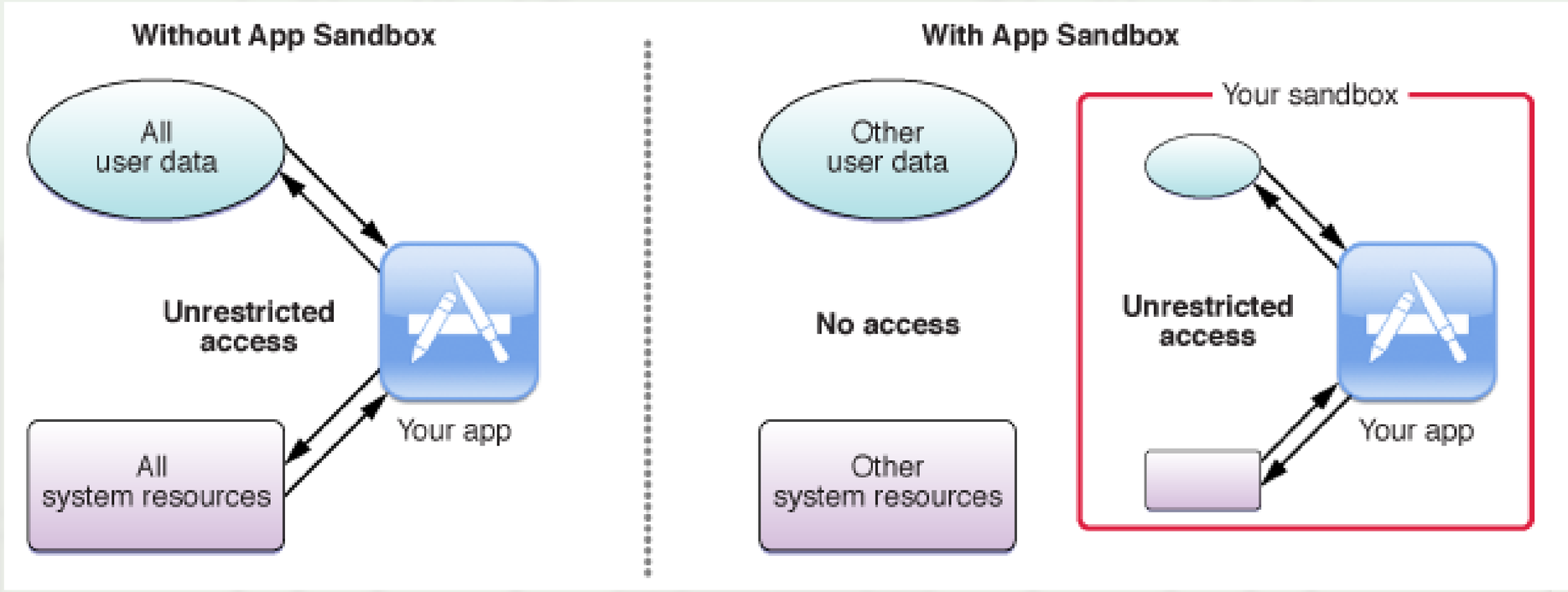
反序列化



沙盒目录文件

沙盒目录

- iOS系统为每个App分配了独立的目录，App只能对自己的目录进行操作，这个目录所在被称为沙盒目录。

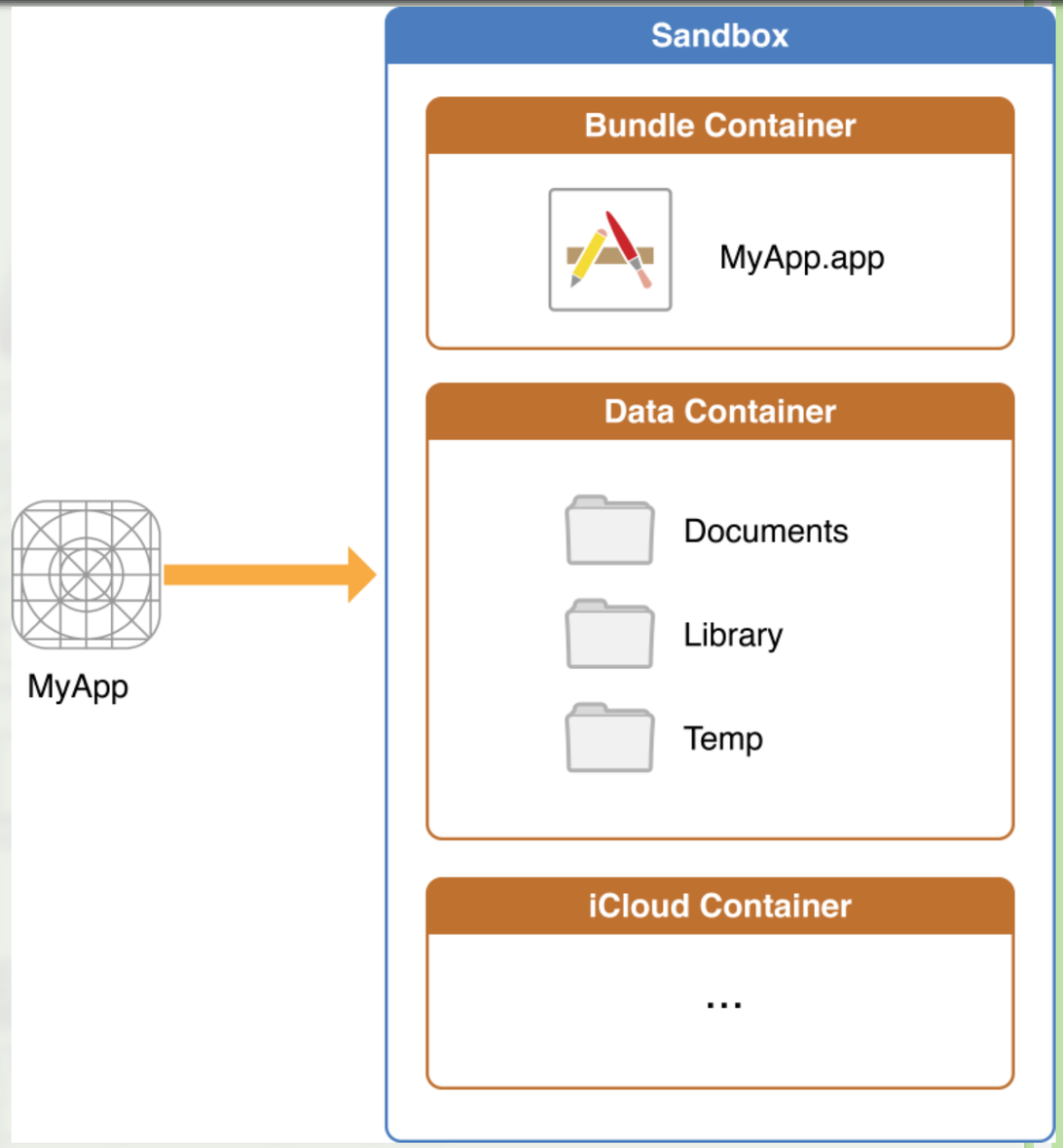


沙盒目录

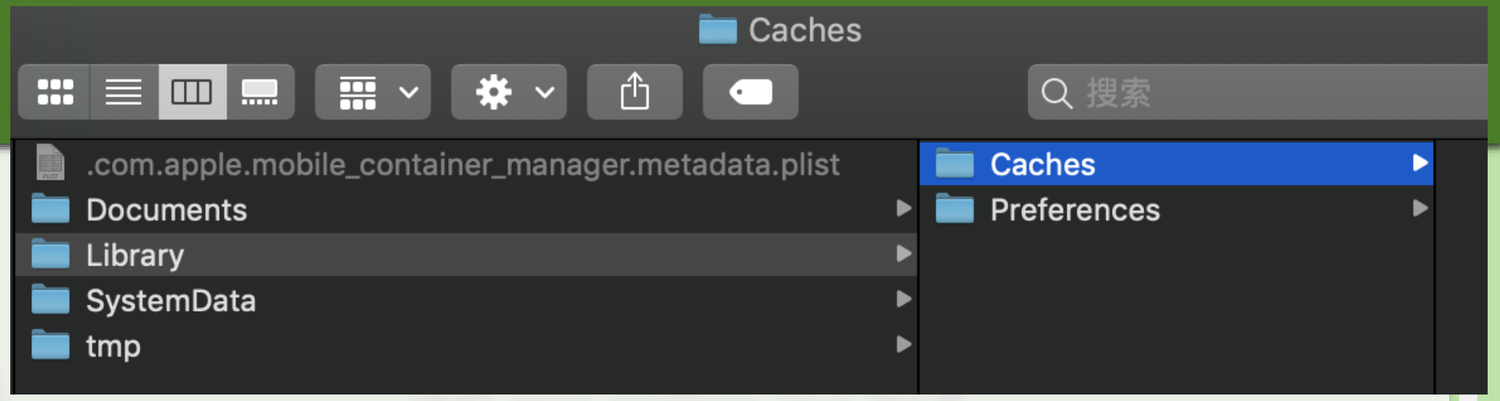
- Documents：保存App运行时生成的需要持久化的数据，iTunes会自动备份该目录；通常用于存放可以对外共享的文件
- Library：保存不对外的数据，除Caches外可被iTunes备份

  Caches：存储临时文件及缓存文件，例如图片、音视频等，空间不足时会被系统删除

  Preferences：存储用户的偏好设置，使用NSUserDefault类以plist的方式存取
- tmp：保存不重要的临时文件，在系统重启后会被清空，不会被iTunes备份。



沙盒目录



- 获取沙盒根目录路径

```
NSString *homeDir = NSHomeDirectory();
```

- 获取Documents目录路径

```
NSString *docDir = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
```

- 获取Library目录路径

```
NSString *libDir = NSSearchPathForDirectoriesInDomains(NSLibraryDirectory, NSUserDomainMask, YES);
```

- 获取Cache目录路径

```
NSString *cacheDir = NSSearchPathForDirectoriesInDomains(NSCachesDirectory, NSUserDomainMask, YES);
```

- 获取tmp目录路径

```
NSString *tmpDir = NSTemporaryDirectory();
```

NSFileManager存取沙盒文件及文件夹

- NSFileManager是沙盒目录的文件管理类，单例

- 通过defaultManager方法获取实例：

```
NSFileManager *fm = [NSFileManager defaultManager];
```

- 判断文件是否存在，并且返回是文件还是文件夹：

```
[fm fileExistsAtPath:filePath isDirectory:&isDirectory];
```

- 遍历文件夹：

```
[fm contentsOfDirectoryAtPath:filePath error:&error];
```

- 复制或者移动文件：

```
[fm copyItemAtPath:sourceFilePath toPath:targetFilePath error:nil];
[fm moveItemAtPath:sourceFilePath toPath:targetFilePath error:nil];
```

- 查阅NSFileManager.h了解更详细的API

NSFileManager文件读写

```
- (IBAction)writeData:(id)sender {
    NSString *document = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES).firstObject;

    NSString *filePath = [document stringByAppendingPathComponent:@"data"];

    Byte byteData[] = {2,3,5,7,11,13,17,19};

    NSData *data = [NSData dataWithBytes:byteData length:8];

    [data writeToFile:filePath atomically:YES];
}

- (IBAction)readData:(id)sender {
    NSString *document = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES).firstObject;

    NSString *filePath = [document stringByAppendingPathComponent:@"data"];

    NSData *data = [NSData dataWithContentsOfFile:filePath];
    NSLog(@"data %@", data);
}
```

NSBundle读取App资源

- Xcode编译App时，会把项目内的图片、xib、音频等资源文件一并打包到.app文件中
- 而使用NSFileManager读取这些资源，需要知道其文件路径参数
- NSBundle就是系统提供，用来辅助读取这些资源的类

```
NSBundle * mainBundle = [NSBundle mainBundle];
```

- 获取mainBundle后，通过mainBundle可以查找对应的资源：

```
NSString *path =[mainBundle pathForImageResource:@"some_pic_name"]; // 获取图片资源路径
```

- 也可以通过mainBundle直接加载xib：

```
[[NSBundle mainBundle] loadNibNamed:@"SSProgressView" owner:self options:nil];
```

- 如何读取CocoaPods安装的Pod库的资源？例如访问BDTestPod的资源：

```
NSString *path = [[NSBundle mainBundle]
pathForResource:@"BDTestPod" ofType:@"bundle"];
NSBundle *podBundle = [NSBundle bundleWithPath:path];
```

NSUserDefaults用户偏好设置存取

- NSUserDefaults用来存储用户设置、系统配置等一些小的数据。

  以键值对key-value的形式保存在沙盒中，是单例的，也是线程安全的

  存储路径为：沙盒目录的Library-->Preferences文件夹中

  因为数据是明文存储在plist文件中，不安全

  即使只修改一个key都会加载整个文件，数据多加载慢(IO、内存)，不适合存储大量数据

- 支持的数据类型有NSString、 NSNumber、NSDate、 NSArray、NSDictionary、BOOL、NSInteger、NSFloat等系统定义的数据类型

  即使对象是NSArray或NSDictionary，他们存储的类型也应该是以上范围包括的。

- 如果要存放其他数据类型或者自定义的对象（如自定义的类对象），则必须将其转换成NSData存储。

  自定义对象可通过实现NSCoding协议，实现对象的序列化和反序列化

- NSUserDefaults会由系统自动将数据写入plist中

  旧版iOS需要调用synchronize方法手动同步写入plist，否则更新数据后系统还没写入时用户退出应用将造成数据丢失

  最新的iOS版本不需要手动同步

NSUserDefaults的读写例子

- 写入key "x" : 1024，key "name": Tom， key "button" : 字典类型对象

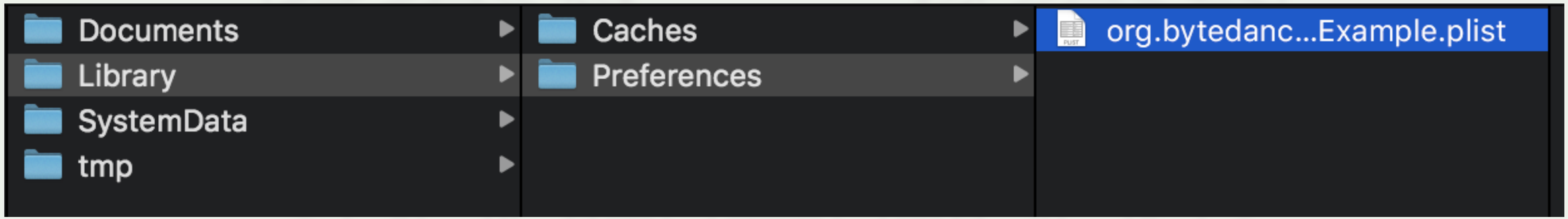
```
[[NSUserDefaults standardUserDefaults] setInteger:1024 forKey:@"x"];
[[NSUserDefaults standardUserDefaults] setObject:@"Tom" forKey:@"name"];
[[NSUserDefaults standardUserDefaults] setObject:@{
                                                   @"text": @"Cancel",
                                                   @"color": @"#ff0000",
                                                  } forKey:@"button"];
```

- 读取key "x", key "name", key "button"

```
NSInteger x = [[NSUserDefaults standardUserDefaults] integerForKey:@"x"];
NSString *name = [[NSUserDefaults standardUserDefaults] stringForKey:@"name"];
NSDictionary *dict = [[NSUserDefaults standardUserDefaults] objectForKey:@"button"];
```

NSUserDefaults的读写例子

- 写入后，沙盒目录将产生相应plist文件（XML格式）



MMKV

- 实际开发中，由于NSUserDefault的性能较差且同步不及时，多用第三方库MMKV取代
- 但是，因为某些系统库仍会读取NSUserDefault上的值，NSUserDefault在工程中仍占有一席之地。
- MMKV

  基于mmap内存映射的移动端通用key-value组件

  底层序列化/反序列化使用protobuf实现，性能高，稳定性强

  开源地址 https://github.com/tencent/mmkv

数据库存储

SQLite3嵌入式关系数据库

- SQLite3是一款轻型的关系型数据库，在移动端中广泛应用。
- SQLite3基于C语言实现，OC可以直接兼容；iOS系统也自带SQLite3，提供的方法可直接操作数据库。
- 创建/打开数据库：

```
NSString *path = [NSHomeDirectory()
stringByAppendingPathComponent:@"test.db"];
sqlite3 *database;
sqlite3_open([path UTF8String], &database);
```

- 建表：

```
const char *createSQL = "create table if not exists
test_table_name(id integer primary key
autoincrement,test_name_key char)";
char *error;
sqlite3_exec(database, createSQL, NULL, NULL, &error);
```

SQLite3

- 执行sql语句：

```
sqlite3_stmt *stmt;
const char *insertSQL = "insert into test_table_name(test_name_key) values('anyname')";
int insertResult = sqlite3_prepare_v2(database, insertSQL, -1, &stmt, nil);
if (insertResult == SQLITE_OK)  sqlite3_step(stmt);
```

- 结束处理：

```
// stmt是中间创建的结果，需要销毁
sqlite3_finalize(stmt);
// 关闭数据库，释放文件句柄等资源
sqlite3_close(database);
```

- sqlite3的原生语言是C语言，接口的调用与OC风格不太一样，感觉较为复杂。

FMDB第三方开源封装

- FMDB对SQLite数据库进行封装，开放OC的接口便于开发者接入，是很普遍使用的iOS第三方数据库。
- 三个核心类：

  1、FMDatabase：表示一个SQLite数据库实例，用于执行sql语句；

  2、FMResultSet：FMDatabase执行查询得到的结果集；

  3、FMDatabaseQueue：多线程用的查询或更新队列；

- 克隆GitHub仓库代码，或可以使用pod引入

  https://github.com/ccgus/fmdb

FMDB的使用

```
// 创建数据库实例
FMDatabase *db = [FMDatabase databaseWithPath:path];
// 打开数据库
[db open]; // open

// 建表
NSString *createSqlStr1 = @"create table if not exits
test_table_name(id integer primary key autoincrement,
test_name_key char)"
[db executeUpdate:CreateSqlStr];

// 插入数据
NSString *insertSqlStr = @"insert into
test_tabl_name(test_name_key) values('anyname')";
[db executeUpdate:insertSqlStr];
```

FMDB

- SQL还可以使用?参数，在执行的时候填写具体的值：

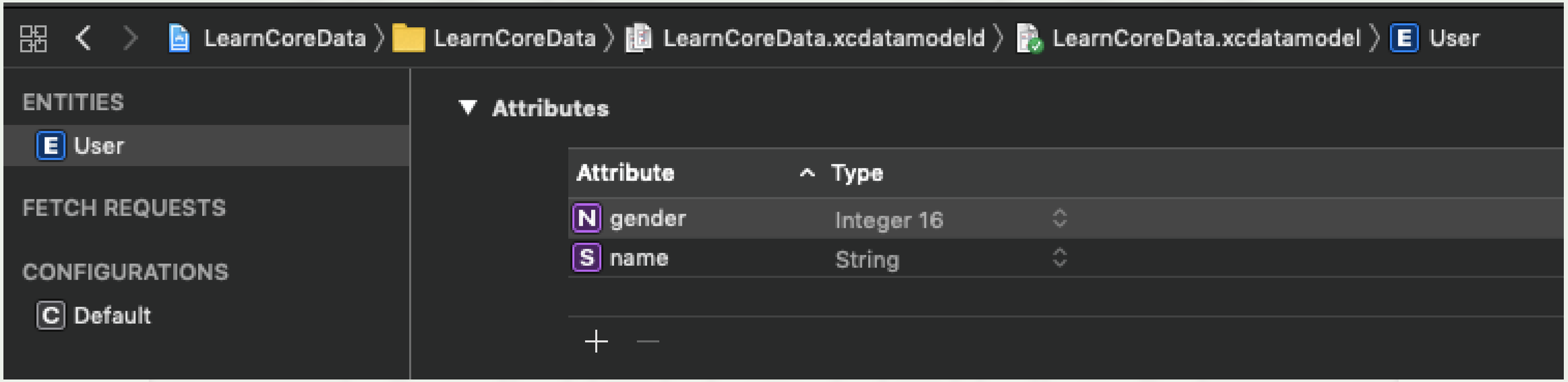
```
NSString *insertSqlStr2 = @"insert into test_table_name(test_name_key) values(?)";
[db executeUpdate:insertSqlStr2, @"test_value"];
```

- 使用FMDatabaseQueue：所有操作都在一个队列中执行，避免多线程操作数据库引起数据异常

```
FMDatabaseQueue *sqlQueue = [FMDatabaseQueue databaseQueueWithPath:path];
[sqlQueue inDatabase:^(FMDatabase * _Nonnull db) {
    NSString *selectSqlStr = @"select id, test_name_key FROM test_table_name";
    FMResultSet *result = [db executeQuery:selectSqlStr]; //执行查询
    while ([result next]) { // 遍历结果集
        int value_id = [result intForColumn:@"id"];
        NSString *value_name = [result stringForColumn:@"test_name_key"];
        NSLog(@"id:%d, name:%@", value_id, value_name);
    }
}];
```

iOS自带数据库CoreData

- iOS系统提供的CoreData框架：接口更加简化，支持可视化操作，对象代码自动生成



- 表结构（可视化操作，自动代码生成）：

```
@interface User (CoreDataProperties)
+ (NSFetchRequest<User *> *)fetchRequest;
@property (nonatomic) int16_t gender;
@property (nullable, nonatomic, copy) NSString *name;
@end
```

CoreData的使用

- 从本地加载对象模型

```
NSString *modelPath = [[NSBundle mainBundle]
      pathForResource:@"LearnCoreData" ofType:@"momd"];
NSManagedObjectModel *model = [[NSManagedObjectModel alloc]
      initWithContentsOfURL:[NSURL fileURLWithPath:modelPath]];
```

- 创建沙盒中的数据库

```
NSPersistentStoreCoordinator *coord =
      [[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:model];
NSString *path = [NShomeDirectory()
stringByAppendingPathComponent:@"database.sqlite"];
[coord addPersistentStoreWithType:NSSqliteStoreType
      configuration:nil URL:[NSURL fileURLWithPath:path] options:nil
error:nil];
```

- 数据库关联缓存

```
NSManagedObjectContext *objContext = [[NSManagedObjectContext alloc]
      initWithConcurrencyType:NSMainQueueConcurrencyType];
objContext.persistentStoreCoordinator = coord;
```

CoreData的使用

- 数据插入

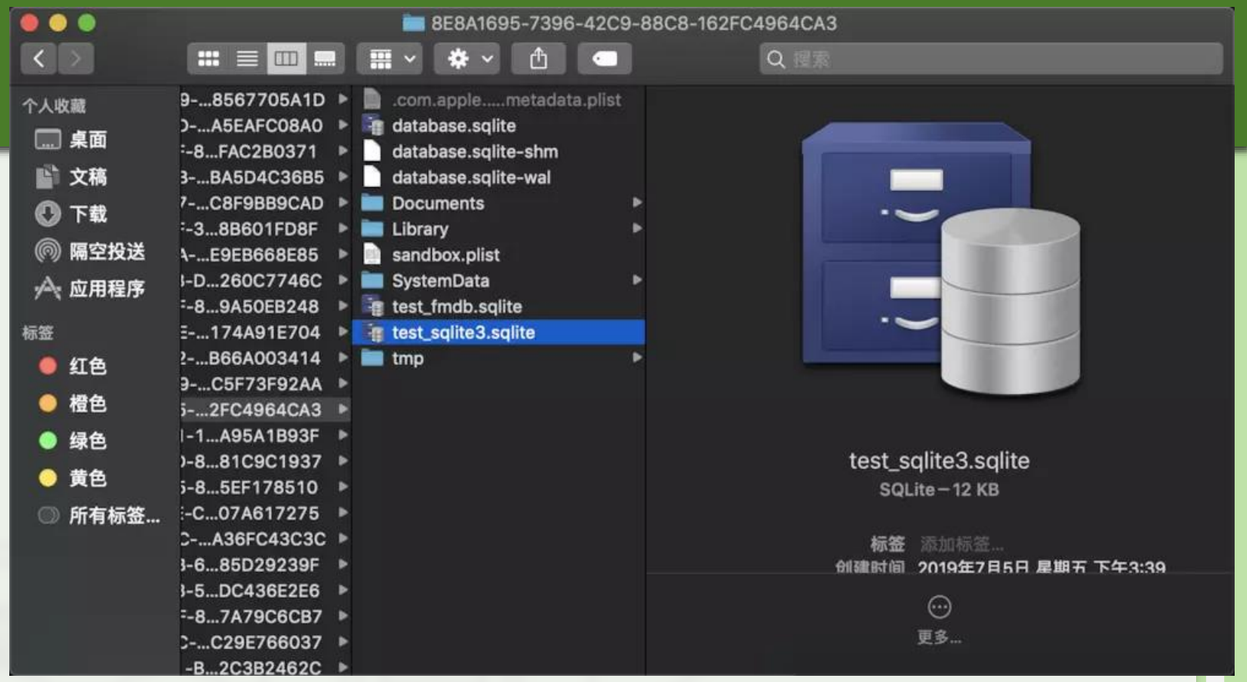
```
User *user = [NSentityDescription insertNewObjectForEntityForName:@"User"
      inManagedObjectContext:objContext];
user.name = [NSString stringWithFormat:@"name_%d", arc4random_uniform(100));
user.gender = arc4random_uniform(2);
NSError *error;
[objContext save:&error];
```

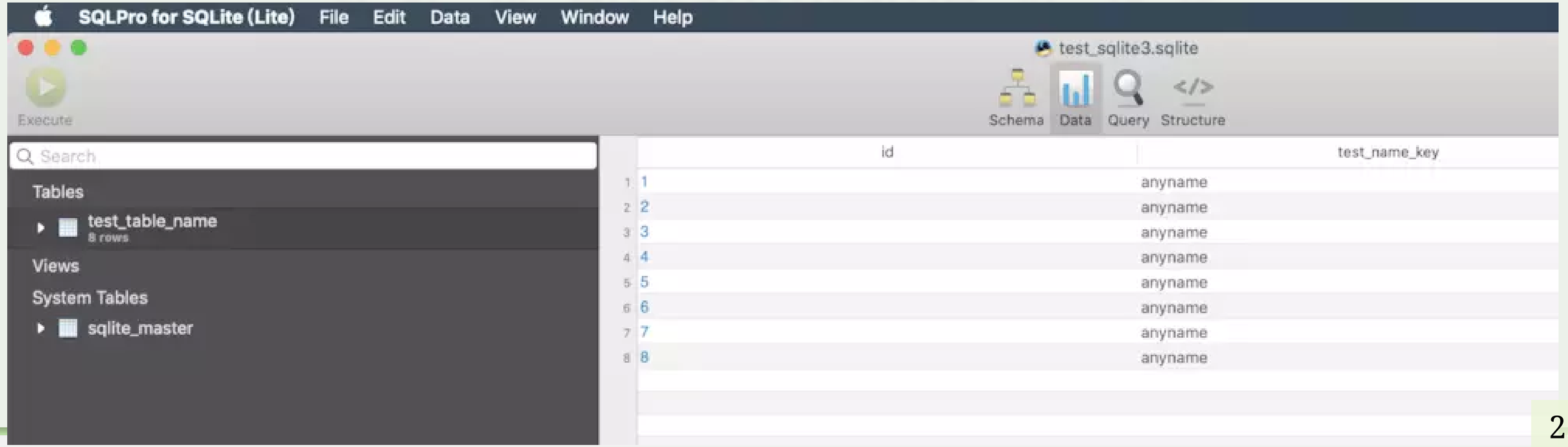
- 数据查询

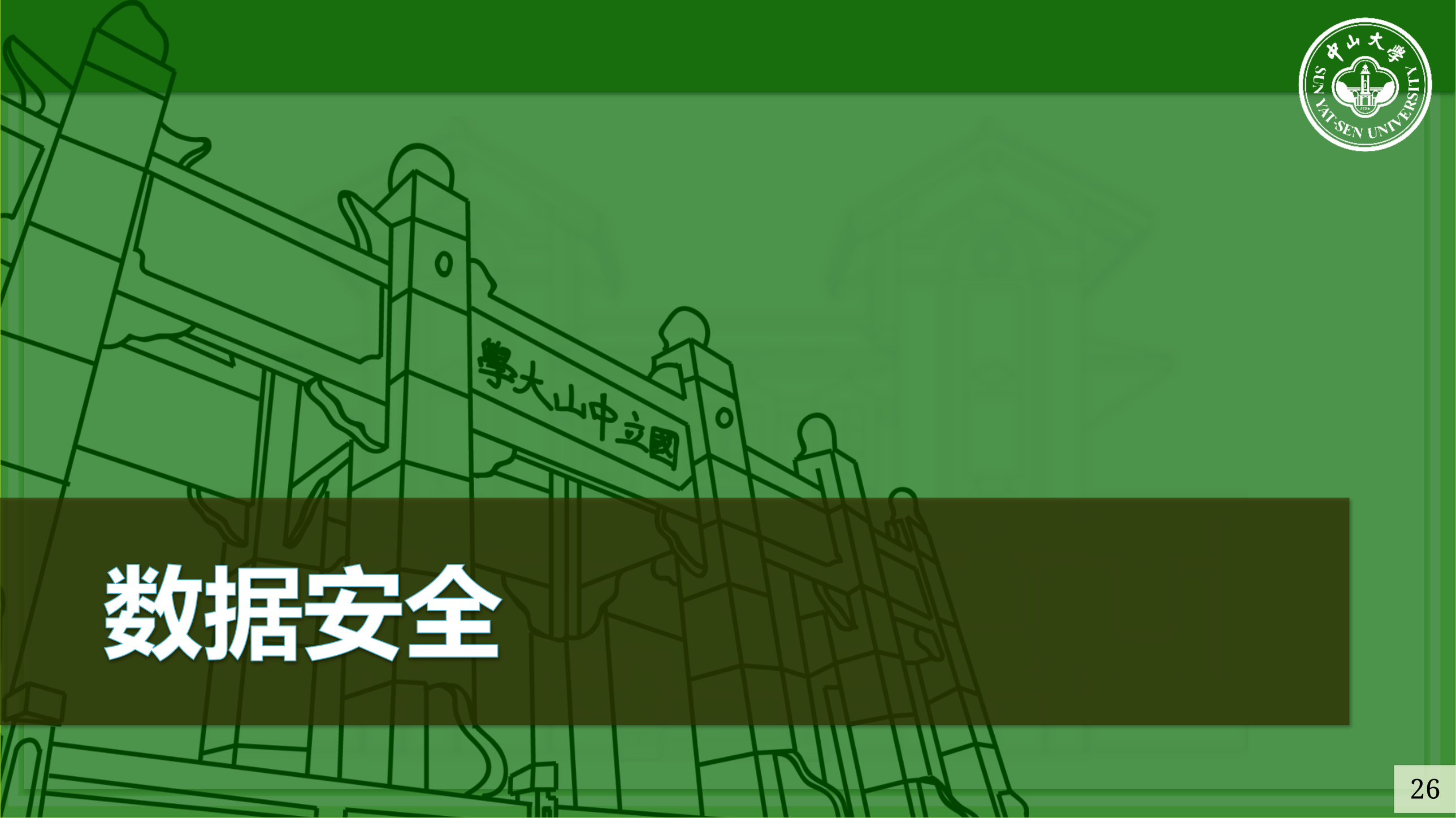
```
NSFetchRequest *fetch = [[NSFetchRequest alloc] initWithEntityName:@"User"];
NSPredicate *predicate = [NSPredicate predicateWithFormate:@"gender=1"];
fetch.predicate = predicate;
NSArray *results = [objContext executeFetchRequest:fetch error:nil];
for (int i = 0; i < results.count; ++i) {
    User *selectedUser = results[i];
    NSLog(@"name_:%@", selectedUser.name);
}
```

数据库文件的调试

- 从沙盒导出项目中实际使用的数据库文件
- 用第三方工具，例如SQLPro for SQLite打开，可以看到里面的数据表及记录







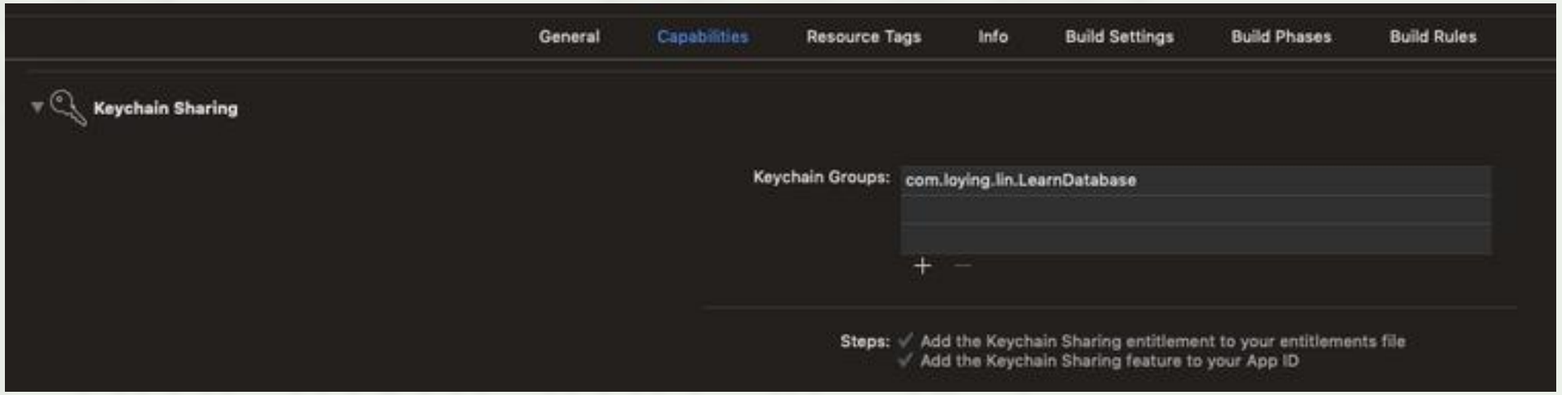
数据安全

Keychain

- 保存在沙盒目录的数据是不安全的，因为用户可以导出沙盒目录的文件进行数据分析。
- Keychain是iOS提供给App存储敏感和安全相关数据用的工具。
- 为保证数据安全，Keychain内的数据是经过加密的。
- Keychain同样可被iTunes备份，即使App重装仍能读取到上次保存的结果。

Keychain的使用

1. 打开Keychain的功能开关(Capabilities)
2. import <Security/Security.h>;
3. 使用API



- 查询select

```
OSStatus SecItemCopyMatching(CFDictionaryRef query, CFTypeRef * result);
```

- 增加add

```
OSStatus SecItemAdd(CFDictionaryRef attributes, CFTypeRef * result);
```

- 更新update

```
OSStatus SecItemUpdate(CFDictionaryRef query, CFDictionaryRef
attributesToUpdate);
```

- 删除delete

```
OSStatus SecItemDelete(CFDictionaryRef query);
```

Keychain的第三方封装

- 官方API非常不友好，幸好苹果官方有提供demo
- 第三方开发者也有人尝试去封装这些接口
- 以KeychainWrapper为例，封装后更简单的接口

```
- (void)savePassword:(NSString *)password;
- (BOOL)deleteItem;
- (NSString *)readPassword;
```

- 比官方更加贴近OC的语法。

KeychainWrapper的使用

- KeychainWrapper的使用样例：

```
KeychainWrapper *wrapper = [[KeychainWrapper alloc]
    initWithService:kKeychainService
            account:self.account
        accessGroup:kKeychainAccessGroup];
NSString *saveStr = [wrapper readPassword];
if (!saveStr) {
    [wrapper savePassword:@"test_password"];
}
NSLog(@"saveStr:%@", saveStr);
```

- 只要保存在keychain，即使应用卸载重装，仍旧能读取到该值。

对象序列化

对象序列化

- 前面介绍了多种文件、键值对、数据库的持久化数据存储方法
- 而内存数据都是以对象的形式存在的，持久化存储前，需要把对象序列化成基本的字节数据类型(NSData)
- 可使用系统提供的NSSecureCoding协议实现对象序列化

  可手动添加字段实现序列化

  可使用Runtime自动实现NSSecureCoding

  可使用成熟的第三方库（例如YYModel）
- 遵守NSSecureCoding协议的类，可以实现这个类到NSData的互相转换

```
@protocol NSSecureCoding
- (void)encodeWithCoder:(NSCoder *)aCoder;
- (nullable instancetype)initWithCoder:(NSCoder *)aDecoder; // NS_DESIGNATED_INITIALIZER
@end
```

- NSSecureCoding协议由iOS6引入，加强安全性

  父类NSCoding不安全的问题，参阅Apple官网文档：***Documentation-Foundation-Archives and Serialization-NSSecureCoding***

NSSecureCoding的使用

- 自定义对象实现协议：（假设有User类，gender、userName是其两个属性）：

```
+ (BOOL)supportsSecureCoding {
    return YES; //支持加密编码
}
// 解码方法（反序列化）
- (instancetype) initWithCoder:(NSCoder *)coder {
    self = [super init];
    self.gender = [[coder decodeObjectForKey:@"gender"] integerValue];
    self.userName = [code decodeObjectOfClass:[NSString class] forKey:@"userName"];
    return self;
}
// 编码方法（序列化）
- (void)encodeWithCoder:(NSCoder *)coder {
    [coder encodeObject:@(self.gender) forKey:@"gender"];
    [coder encodeObject:self.userName forKey:@"userName"];
}
```

- 使用场景

NSUserDefaults：将自定义的对象转换成NSData实例，然后存储到偏好存储中。

文件存储：NSData类提供了一个方法，可以直接将NSData实例存储到文件中

NSSecureCoding的完整例子

- 自定义对象User，声明实现NSSecureCoding协议

```
@interface User : NSObject <NSSecureCoding>

@property (nonatomic, assign) NSInteger userId;
@property (nonatomic, strong) NSString *nickname;
@property (nonatomic, assign) BOOL isVip;

@end
```

NSSecureCoding的完整例子

- 协议的实现

```
- (instancetype)initWithCoder:(NSCoder *)coder
{
    self = [super init];
    if (self) {
        self.userId = [coder decodeIntegerForKey:@"id"];
        self.nickname = [coder decodeObjectForKey:@"name"];
        self.isVip = [coder decodeBoolForKey:@"isVip"];
    }
    return self;
}

- (void)encodeWithCoder:(NSCoder *)coder
{
    [coder encodeInteger:self.userId forKey:@"id"];
    [coder encodeObject:self.nickname forKey:@"name"];
    [coder encodeBool:self.isVip forKey:@"isVip"];
}

+ (BOOL)supportsSecureCoding {
    return YES;
}
```

NSSecureCoding的完整例子

- 写入NSUserDefaults前，自动序列化为NSData

```
User *user = [[User alloc] init];
user.userId = 123;
user.nickname = @"Tom";
user.isVip = YES;

NSData *data = [NSKeyedArchiver archivedDataWithRootObject:user
                                     requiringSecureCoding:YES
                                                     error:nil];
[[NSUserDefaults standardUserDefaults] setObject:data
                                          forKey:@"user"];
```

NSSecureCoding的完整例子

- 读取NSUserDefaults时，得到NSData，反序列化为自定义对象

```
NSData *data = [[NSUserDefaults standardUserDefaults] dataForKey:@"user"];
User *user = [NSKeyedUnarchiver unarchivedObjectOfClass:[User class]
                                               fromData:data
                                                  error:nil];
```

- 文件及数据库的读写与NSUserDefaults类似。

YYModel对象序列化

- YYModel：High performance model framework for iOS/OSX (It's a component of YYKit)

  1、利用iOS的Runtime特点，无需继承；
  2、安全转换数据类型，常见Crash都进行了保护；
  3、扩展性强，提供多种容器扩展；

- YYModel可以使用CocoaPod安装

```
Add pod 'YYModel' to your Podfile
Run pod install or pod update
Import <YYModel/YYModel.h>
```

- 将字典转换为对象：

```
NSDictionary *dict = @{
                       @"gender":@0,
                       @"userName":@"test_name"
                     };
User *user = [User modelWithDictionary:dict];
```

- YYModel还提供丰富的特性，比如说自定义属性名映射、容易类型转换、自定义类的数据映射。

  https://github.com/ibireme/YYModel

YYModel常用方法

字典转模型 + (nullable instancetype)modelWithDictionary:(NSDictionary *)dictionary;

json转模型 + (nullable instancetype)modelWithJSON:(id)json;

模型转NSObject - (nullable id)modelToJSONObject;

模型转NSData - (nullable NSData *)modelToJSONData;

模型转json字符串 - (nullable NSString *)modelToJSONString;

模型深拷贝 - (nullable id)modelCopy;

判断模型是否相等 - (BOOL)modelIsEqual:(id)model;

属性数据映射，用来定义多样化数据时转换声明

+ (nullable NSDictionary<NSString *, id> *)modelCustomPropertyMapper;

属性自定义类映射，用来实现自定义类的转换声明

+ (nullable NSDictionary<NSString *, id> *)modelContainerPropertyGenericClass;

属性黑名单，该名单属性不转换为model

+ (nullable NSArray<NSString *> *)modelPropertyBlacklist;

属性白名单，只有该名单的属性转换为model

+ (nullable NSArray<NSString *> *)modelPropertyWhitelist;

自定义转换，JSON转为Model完成后调用，返回false该model会被忽略

- (BOOL)modelCustomTransformFromDictionary:(NSDictionary *)dic;

自定义转换，Model 转为 JSON 完成后调用，返回false该model会被忽略；

- (BOOL)modelCustomTransformToDictionary:(NSMutableDictionary *)dic

总结

- iOS的本地数据存取，其实就是内存数据的序列化和反序列化。
- 用户数据都会保存在沙盒目录中，读取时可直接指定路径，也可用NSFileManager查找和遍历目录；App资源文件则保存在应用目录，需要用NSBundle读取。
- App在运行过程中，有时需临时保存一些变量，在下次运行时读取，此时可以用轻量级的持久化工具NSUserDefaults；如果数据量较大则需考虑使用数据库进行存储。SQLite3是iOS中最常用的数据库，通常我们会第三方封装库FMDB来操作，简化代码工作量。
- 如果涉及到安全相关的敏感数据，则不应该明文保存在文件、数据库等可以被抓取的地方。此时可以使用iOS提供的Keychain保存，数据经过加密处理，具有较高的安全性。
- 在将对象转换成二进制数据，以及将二进制数据转换成对象时，可以使用系统提供的NSSecureCoding协议，也可以使用第三方库YYModel.

Thanks