本地 sqlite 数据库框架搭建过程

1. 简介：每个应用初期姑且不说，后期肯定会有本地数据库的创建需求，怎么说呢，算是为了更好的用户体验，或着为了少请求接口避免拿重复数据(比如设置更新参数需要更新接口才给数据，否则使用本地数据库内的数据)吧，本人近期就遇到了这个方面的需求，所以对数据库 sqlite 框架进行了搭建，当然我是在著名第三方 FMDB 数据库管理库的基础上进行搭建的。目的是做到最终接口易使用，代码简洁易读，还有需要提供一套数据库迁移逻辑，等等。具体搭建过程如下：

2. 数据库管理框架的搭建

   (1) 数据库管理单例基类MYBaseSQLiteManager的介绍

   以下是头文件

```
#import <Foundation/Foundation.h>
#import "FMDB.h"
/**
 * FMDB管理模型基类
 *
 * @note  提供数据库底层增删改查等操作。 ps.本类为基类，尽量别含业务逻辑
 * @note  子类可以是具体的数据库操作类，可以含业务逻辑处理。
 */


@interface MYBaseSQLiteManager : NSObject

@property (strong, nonatomic) FMDatabase *database;

//获取单例对象
+ (instancetype)defaultManager;


//获取单例对象
+ (instancetype)defaultManager;

/**
 * 创建数据表
 *
 * @ param sql 数据库查询语句 eg. [NSString stringWithFormat:@"create table if not exists %@ (ID integer primary key autoincrement, name text, phone text)", T_Person];
 */
- (BOOL)createTableUsingSQL:(NSString *)sql;

/**
 * 数据插入
 *
 * @ param sql 数据库查询语句 eg. [NSString stringWithFormat:@"insert into %@ (name, phone) values (?, ?)", T_Person]
 * @ param arguments 查询语句参数 eg. @[]
 */
- (BOOL)insertObjectUsingSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments;
```

```
/**
 * 数据删除
 *
 * @ param sql 数据库查询语句 eg. [NSString stringWithFormat:@"delete from %@ where name=?", T_Person]
 * @ param arguments 查询语句参数 eg. @[]
 */
- (BOOL)removeObjectUsintSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments;

/**
 * 数据修改
 *
 * @ param sql 数据库查询语句 eg. [NSString stringWithFormat:@"delete from %@ where name=?", T_Person]
 * @ param arguments 查询语句参数 eg. @[]
 */
- (BOOL)updateObjectUsingSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments;

/**
 * 数据查询
 *
 * @ note 记得查询结果FMResultSet操作结束后关闭数据库，提前关闭会导致后续操作出现问题
 * @ param sql 数据库查询语句 eg. [NSString stringWithFormat:@"select *from %@", T_Person]
 * @ param arguments 查询语句参数 eg. @[]
 */
- (FMResultSet *)queryObjectUsingSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments;

/**
 * 关闭数据库
 *
 * @ note 数据库查询结果FMResultSet操作结束后调用
 */
- (void)closeDatabaseAfterQueryDown;

/**
 * 查询数据库中是否存在某张表
 *
 * @ param tableName  列表名，如T_Person
 */
- (BOOL)ifTableExists:(NSString *)tableName;


/**
 * 查询沙盒中是否存在数据库
 *
 * @ param tableName  列表名，如T_Person
 */
- (BOOL)ifSQLiteFileExists;

/**
 * 数据库新增字段
 *
 * @ param sql 数据库查询语句 eg.
 * @ param arguments 查询语句参数 eg. @[]
 * @ note sqlite不允许这种在数据库表中删除列的方式 (DROP COLUMN column_name)。这种修改字段类型的方式(ALTER COLUMN column_name type)
 * @ note SQLite 仅仅支持 ALTER TABLE 语句的一部分功能，我们可以用 ALTER TABLE 语句来更改一个表的名字，也可向表中增加一个字段（列），但是我们不能删除一个已经存在的字段，或者更改
   一个已经存在的字段的名称、数据类型、限定符
 */
- (BOOL)alterTableUsingSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments;

@end
```

先说下基类的封装目的：首先，这个类提供数据库底层的增删改查等操作，内部含database的访问准备逻辑，比如操作前先打开db，使用结束后关闭db等等。后面我们会创建一个子类，是具体的数据库操作类，含业务逻辑的具体处理，当然，子类执行具体增删改查时需要调用父类的相应方法，尽管可能最终FMDB的sql语法是一样的，我们也要作区分，因为便于代码的阅读(比如：insert into或update set

操作其实对FMDB来说都是执行update表语法，但我们基类做了区分，我们就很清楚，我们代码的操作意图，是要插入数据，还是修改数据）

以下是实现文件的代码逻辑：

```
//数据库
static const NSString *kDatabaseName = @"GuDaShi";

@interface MYBaseSQLiteManager ()

@end

@implementation MYBaseSQLiteManager

+ (instancetype)defaultManager{
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [[self alloc] init];
        if(!instance.database){
            NSString *databasePath = [instance filePath];
            instance.database = [FMDatabase databaseWithPath:databasePath];
            if(!instance.database){
                NSAssert(NO, instance.database.lastErrorMessage);
            }
        }
    });

    //打开数据库
    [instance openDatabase];
    //设置缓冲 提供效率
    [instance.database setShouldCacheStatements:YES];
    return instance;
}

+ (instancetype)allocWithZone:(struct _NSZone *)zone{
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        instance = [super allocWithZone:zone];
    });
    return instance;
}
```

我们一步一步来解说实现文件：

这部分代码是单例类的创建，这里有个需要注意的事项，就是单例怎么继承？因为我们平时继承用的都是普通类，单例类，因为我们知道只有一个实例，那么如果编写不当的话，可能实例出来的并不是子类，那后续就不能执行子类具体数据库的管理操作了。所以单例创建关键代码在于：

```
instance = [[self alloc] init];
```

这个是正确的操作，也就是说

在defaultManager里，如果我们不用self，用的是MYBaseSQLiteManager来alloc，

那么这个单例就不能被继承！

再接下来，就是具体的数据库操作逻辑了，我们一个一个具体操作来看：

1)创建一张表的操作如下：

```
#pragma mark public methods

- (BOOL)createTableUsingSQL:(NSString *)sql{
    if(!self.database)
        return NO;

    [self openDatabase];
    BOOL isDown= [_database executeStatements:sql];
    [_database close];
    if(!isDown){
        NSLog(@"数据表创建失败，原因：%@", _database.lastErrorMessage);
    }else{
        //          NSLog(@"数据表创建成功");
    }
    return isDown;
}
```

具体sql示例：

[NSString stringWithFormat:@"create table if not exists %@ (id integer primary key

autoincrement,name text,lastedittime text,userid text)", T_N_DataUpdateInfo]

2)插入一条数据

```
- (BOOL)insertObjectUsingSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments{
    if(!self.database)
        return NO;

    [self openDatabase];
    BOOL isDown = [_database executeUpdate:sql withArgumentsInArray:arguments];
    [_database close];
    if(!isDown){
        NSLog(@"数据插入失败，原因：%@", _database.lastErrorMessage);
        return NO;
    }
    //          NSLog(@"数据插入成功");
    return YES;
}
```

具体sql示例：

[NSString stringWithFormat:@"insert into %@(name,lasteditime) values (?,?)",

T_N_DataUpdateInfo] withArgumentsArray:@[name, lasteditime]

3)删除一条数据

```
- (BOOL)removeObjectUsintSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments{
    if(!self.database)
        return NO;

    [self openDatabase];
    BOOL isDown = [_database executeUpdate:sql withArgumentsInArray:arguments];
    [_database close];
    if(!isDown){
        NSLog(@"数据删除失败，原因：%@", _database.lastErrorMessage);
        return NO;
    }
    //        NSLog(@"数据删除成功");
    return YES;
}
```

具体sql示例：

[NSString stringWithFormat:@"delete from %@ where name=?",

T_N_DataUpdateInfo] withArgumentsArray:@[name]

4)更新一条数据

```
- (BOOL)updateObjectUsingSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments{
    if(!self.database)
        return NO;

    [self openDatabase];
    BOOL isDown = [_database executeUpdate:sql withArgumentsInArray:arguments];
    [_database close];
    if(!isDown){
        NSLog(@"数据更新失败，原因：%@", _database.lastErrorMessage);
        return NO;
    }
    //        NSLog(@"数据更新成功");
    return YES;
}
```

具体sql示例：

[NSString stringWithFormat:@"update %@ set lasteditime=? where name=?",

T_N_DataUpdateInfo] withArgumentsArray:@[lasteditime, name]

5)查询数据

```
- (FMResultSet *)queryObjectUsingSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments{
    if(!self.database)
        return nil;

    [self openDatabase];
    FMResultSet *resultSet = [_database executeQuery:sql withArgumentsInArray:arguments];
    return resultSet;
}

- (void)closeDatabaseAfterQueryDown{
    if(!self.database)
        return;

    [_database close];
}
```

具体sql示例：

[NSString stringWithFormat:@"select * from %@ where name=? and userid=?",

T_N_DataUpdateInfo] withArgumentsArray:@[name, userid]

注意，这里，我们有个closeDatabaseAfterQueryDown方法，是单独出来的，因为查询操作，如果提前把数据库关闭了，会导致返回的FMResultSet内存被回收，也就是外界处理时候会出错。因此，暂时给了一个单独的接口，让外界对FMResultSet处理完后进行数据库的关闭操作

6)查询数据库表是否存在

```
- (BOOL)ifTableExists:(NSString *)tableName{
    if(!self.database)
        return NO;

    [self openDatabase];

    FMResultSet *rs = [self.database executeQuery:@"select count(*) as 'count' from sqlite_master where type ='table' and name = ?", tableName];
    while ([rs next])
    {
        // just print out what we've got in a number of formats.
        NSInteger count = [rs intForColumn:@"count"];
        if (0 == count){
            [_database close];
            return NO;
        }else{
            [_database close];
            return YES;
        }
    }
    [_database close];
    return NO;
}
```

7)查询数据库是否存在

```
- (BOOL)ifSQLiteFileExists{
    BOOL flag = [[NSFileManager defaultManager] fileExistsAtPath:[self filePath] isDirectory:nil];
    return flag;
}
```

8)数据库字段添加操作

```
- (BOOL)alterTableUsingSQL:(NSString *)sql withArgumentsArray:(NSArray *)arguments{
    if(!self.database)
        return NO;

    [self openDatabase];
    BOOL isDown = [_database executeUpdate:sql withArgumentsInArray:arguments];
    [_database close];
    if(!isDown){
        NSLog(@"数据库新增字段失败，原因：%@", _database.lastErrorMessage);
        return NO;
    }
    //          NSLog(@"数据更新成功");
    return YES;
}
```

具体sql示例：

[NSString stringWithFormat:@"alter table %@ add roomid text default %@",

T_N_GaoShouZiXuan, @"0"] withArgumentsArray:@[]

9)接下来是些其他操作，就不多作说明了

```
#pragma mark Private Methods

- (NSString *)filePath{
   NSString *path = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES) firstObject];
   return  [path stringByAppendingPathComponent: [NSString stringWithFormat:@"%@.sqlite", kDatabaseName]];
}

- (BOOL)openDatabase{
   if (![_database open]) { //打开失败
      [_database close];
      NSAssert(NO, @"数据库打开失败");
      return NO;
   }
   return YES;
}

- (BOOL)closeDatabase{
   if (![_database close]) {
      NSAssert(NO, @"数据库关闭失败");
      return NO;
   }
   return YES;
}
```

⑵数据库管理单例子类MYSQLiteManager的介绍

接下来就是我们的重头戏了，含业务逻辑处理的项目具体数据库管理类的介绍

同样的，我们先看看头文件的内容，对整个类封装有个大概的了解

```
#import "MYBaseSQLiteManager.h"
#import <CoreGraphics/CoreGraphics.h>
/**
 * FMDB管理模型
 *
 * 本类继承自MYBaseSQLiteManager，提供具体列表创建和相关业务逻辑处理
 */

//当前数据库版本，从1开始
#define kSQLite_Version 3

//列表名
extern NSString *const T_N_DataUpdateInfo;//数据更新信息列表
extern NSString *const T_N_GaoShouZiXuan;//高手自选列表

//字段值
extern NSString *const F_V_GaoShouZiXuan;//T_N_DataUpdateInfo表中name值："高手自选列表"，目的：获取自选股列表上一次编辑时间
extern NSString *const F_V_TouZiCeLve;//T_N_DataUpdateInfo表中name值："投资策略"，目的：获取投资策略上一次浏览时间
extern NSString *const F_V_ZhiBoTiXing;//T_N_DataUpdateInfo表中name值："直播提醒"，目的：获取直播提醒上一次浏览时间
extern NSString *const F_V_GuShiPingLun;//T_N_DataUpdateInfo表中name值："股市评论"，目的：获取股市评论上一次浏览时间
extern NSString *const F_V_UserLogin;//T_N_DataUpdateInfo表中name值："用户登录"，目的：获取用户登录时间


@interface MYSQLiteManager : MYBaseSQLiteManager

- (void)migrationsLocalSQLiteIfNeeded;//数据库迁移，完成迁移后会自动更新[MYUserDefaults standardUserDefaults].current_sqlite_version；注意每个版本数据库结构改变的话，具体增删改查操作要改掉，
  并且确保迁移操作中也有相应结构的修改逻辑

//-------------------------------------------------数据更新信息列表-------------------------------------------------
- (BOOL)createTableUpdateInfoIfNeeded;//创建数据更新信息列表

- (BOOL)insertIntoUpdateInfoWithName:(NSString *)name lastEdittime:(NSString *)lasteditime;//插入一条数据更新信息
- (BOOL)insertIntoUpdateInfoWithName:(NSString *)name lastEdittime:(NSString *)lasteditime userid:(NSString *)userid;//插入某个用户一条数据更新信息

- (BOOL)upDataUpdateInfoName:(NSString *)name setLastEdittime:(NSString *)lasteditime;//修改一条更新信息的最近一次更新时间字段
- (BOOL)upDataUpdateInfoName:(NSString *)name setLastEdittime:(NSString *)lasteditime userid:(NSString *)userid;//修改某个用户一条更新信息的最近一次更新时间字段

- (BOOL)deleteUpdateInfoName:(NSString *)name;//删除某条更新时间信息
- (BOOL)deleteUpdateInfoName:(NSString *)name userid:(NSString *)userid;//删除某个用户某条更新时间信息

- (NSString *)getLastEdittimeOfName:(NSString *)name;//获取某条更新信息的最近一次更新时间字段
- (NSString *)getLastEdittimeOfName:(NSString *)name userid:(NSString *)userid;//获取某个用户某条更新信息的最近一次更新时间字段


//-------------------------------------------------高手自选列表-------------------------------------------------
- (BOOL)createTableGaoShouZiXuanIfNeeded;//创建高手自选列表
- (BOOL)insertIntoGaoShouZiXuanWithImageurl1:(NSString *)imageurl1 imageurl2:(NSString *)imageurl2 useriddes:(NSString *)useriddes title:(NSString *)title roomid:(NSString *)roomid selfstocksuper_id:(NSString *)
  selfstocksuper_id;//插入一条自选高手信息
- (BOOL)deleteAllTableGaoShouZiXuan;//删除所有自选高手信息
- (NSMutableArray *)selectAllGaoShouZiXuan;//查询所有自选高手信息
```

可以看到，我们这个类，主要进行项目中用到的所有表的创建及管理逻辑，数据获取及更新等。一开始数据库版本，这个是用来做数据库迁移用的；接下来是表名和字段值的定义，没看到值是不是，没错，这种写法定义在头文件，具体赋值在实现文件，这些规范对后面的数据库管理很有帮助，我们很清楚地知道我们当前数据库有哪些表，我们查询的哪些字段值是有特殊意义需要作说明的等；再来是数据库迁移逻辑，内部是根据存在本地的数据库版本号和我们一开始定义的当前数据库版本号进行判断对于更新app的用户第一次打开我们app需不需要进行

数据库迁移操作；再接下来，就是各张表的操作逻辑了，传入参数可以是一个model，传出参数也可以是一个model或model的数组，这个可以很灵活。

以下是实现文件的代码逻辑：

```
#import "MYSQLiteManager.h"
#import "GuPiaoDaiMaiName.h"
#import "MYUserDefaults.h"
#import "GaoShouZiXuanDataModel.h"

//数据表
NSString *const T_N_DataUpdateInfo = @"T_DataUpdateInfo";
NSString *const T_N_GaoShouZiXuan= @"T_GaoShouZiXuan";

//字段
NSString *const F_V_GaoShouZiXuan = @"gaoShouZiXuan";
NSString *const F_V_TouZiCeLve = @"touZiCeLve";
NSString *const F_V_ZhiBoTiXing = @"zhiBoTiXing";
NSString *const F_V_GuShiPingLun = @"guShiPingLun";
NSString *const F_V_UserLogin = @"userLogin";


@implementation MYSQLiteManager

- (void)migrationsLocalSQLiteIfNeeded{
    //初始化所有列表
    if([self ifTableExists:T_N_DataUpdateInfo] == NO){//初始化数据更新信息表
        [self createTableUpdateInfoIfNeeded];//
    }

    if([self ifTableExists:T_N_GaoShouZiXuan] == NO){
        [self createTableGaoShouZiXuanIfNeeded];//初始化高手自选表
    }


    //如果列表结构有变动，则进行数据迁移
    CGFloat current_sqlite_version = [[MYUserDefaults standardUserDefaults] current_sqlite_version];
    if(current_sqlite_version == 0 || current_sqlite_version == kSQLite_Version){//不需重复进行数据库迁移
        [[MYUserDefaults standardUserDefaults] setCurrent_sqlite_version:kSQLite_Version];
    }else{//数据库已更新，进行数据库迁移
        [self migrations:current_sqlite_version];
    }
}
```

同样的，我们一步一步来解说实现文件：

首先是数据库迁移逻辑，我们先进行了一次数据表的初始化工作，确保所有表都存在，然后比对本地数据库版本和当前数据库版本，如果本地数据库版本号为0(说明第一次安装)，或者数据库版本等于当前数据库版本号(已经做过数据库迁移)，那么因为我们已经创建好了所有的表了或者已经执行完所有的数据库迁移操作了，所以不需要进行数据库迁移工作。如果本地数据库版本号小于当前数据库版本号，那么我们就需要进行一次数据库迁移，因为对于这种客户app的数据库是

旧的，难保里面数据已经存在但表结构在新版本中发生了变化，不更新会导致数据库表读写异常。具体的迁移逻辑如下：

```
- (void)migrations:(NSInteger)oldVersion{
    if(oldVersion == kSQLite_Version || oldVersion > kSQLite_Version){//递归调用临界点，确保数据库迁移完毕(从某个低版本，迁移至最新版本)
        [MYUserDefaults standardUserDefaults].current_sqlite_version = kSQLite_Version;
        return;
    }

    switch (oldVersion) {
        case 0:{
            //数据库版本从1开始，不可能为0
        }
            break;
        case 1:{//数据库版本1 -> 数据库版本2，由v1.7.0 -> v1.7.1，(1)我们往高手自选列表新增了字段userid字段；(2)同时需要删除师自选表内的数据；(3)同时我们往信息更新表新增字段userid
            //(1)
            [self alterTableUsingSQL:[NSString stringWithFormat:@"alter table %@ add roomid text default %@", T_N_GaoShouZiXuan, @"0"] withArgumentsArray:@[]];//本地自选股列表添加昨收价格字段
            //(2)
            [self deleteAllTableGaoShouZiXuan];
            [self deleteUpdateInfoName:F_V_GaoShouZiXuan];
            //(3)
            [self alterTableUsingSQL:[NSString stringWithFormat:@"alter table %@ add userid text default %@", T_N_DataUpdateInfo, nil] withArgumentsArray:@[]];
        }
            break;
        case 2:{//数据库版本2 -> 数据库版本3，由v1.7.1 -> v1.7.2  (1)我们往高手自选列表新增了字段selfstocksuper_id; (2)同时需要删除师自选表内的数据
            //(1)
            [self alterTableUsingSQL:[NSString stringWithFormat:@"alter table %@ add selfstocksuper_id text default %@", T_N_GaoShouZiXuan, nil] withArgumentsArray:@[]];
            //(2)
            [self deleteAllTableGaoShouZiXuan];
            [self deleteUpdateInfoName:F_V_GaoShouZiXuan];
        }
            break;
        default:
            break;
    }

    oldVersion ++;
    [self migrations:oldVersion];
}
```

递归调用，让数据库版本逐级升级到最新。

再接下来，就是具体表的操作逻辑了，以下就以T_N_DataUpdateInfo数据更新表为例子进行讲解，其他表逻辑类似

1) T_N_DataUpdateInfo表创建

```
- (BOOL)createTableUpdateInfoIfNeeded{
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == NO)
        return [self createTableUsingSQL:[NSString stringWithFormat:@"create table if not exists %@ (id integer primary key autoincrement,name text,lasteditime text,userid text)", T_N_DataUpdateInfo]];
    return NO;
}
```

2) T_N_DataUpdateInfo表插入

```
- (BOOL)insertIntoUpdateInfoWithName:(NSString *)name lastEdittime:(NSString *)lasteditime{
    name = name == nil ? @"" : name;
    lasteditime = lasteditime == nil ? @"" : lasteditime;
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == YES){
        return [self insertObjectUsingSQL:[NSString stringWithFormat:@"insert into %@(name,lasteditime) values (?,?)", T_N_DataUpdateInfo] withArgumentsArray:@[name, lasteditime]];
    }
    return NO;
}
```

```
- (BOOL)insertIntoUpdateInfoWithName:(NSString *)name lastEdittime:(NSString *)lastedittime userid:(NSString *)userid{
    name = name == nil ? @"" : name;
    lastedittime = lastedittime == nil ? @"" : lastedittime;
    userid = userid == nil ? @"": userid;
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == YES){
        return [self insertObjectUsingSQL:[NSString stringWithFormat:@"insert into %@(name,lastedittime,userid) values (?,?,?)", T_N_DataUpdateInfo] withArgumentsArray:@[name, lastedittime, userid]];
    }
    return NO;
}
```

注意：插入数据时需对参数合法性进行判断，确保数据库插入不会出现异常。

3) T_N_DataUpdateInfo表更新

```
- (BOOL)upDataUpdateInfoName:(NSString *)name setLastEdittime:(NSString *)lastedittime{
    name = name == nil ? @"" : name;
    lastedittime = lastedittime == nil ? @"" : lastedittime;
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == YES){
        return [self updateObjectUsingSQL:[NSString stringWithFormat:@"update %@ set lastedittime=? where name=?", T_N_DataUpdateInfo] withArgumentsArray:@[lastedittime, name]];
    }
    return NO;
}
- (BOOL)upDataUpdateInfoName:(NSString *)name setLastEdittime:(NSString *)lastedittime userid:(NSString *)userid{
    userid = userid == nil ? @"" : userid;
    name = name == nil ? @"" : name;
    lastedittime = lastedittime == nil ? @"" : lastedittime;
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == YES){
        return [self updateObjectUsingSQL:[NSString stringWithFormat:@"update %@ set lastedittime=? where name=? and userid=?", T_N_DataUpdateInfo] withArgumentsArray:@[lastedittime, name, userid]];
    }
    return NO;
}
```

4) T_N_DataUpdateInfo表删除

```
- (BOOL)deleteUpdateInfoName:(NSString *)name{
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == YES){
        return [self updateObjectUsingSQL:[NSString stringWithFormat:@"delete from %@ where name=?", T_N_DataUpdateInfo] withArgumentsArray:@[name]];
    }
    return NO;
}
- (BOOL)deleteUpdateInfoName:(NSString *)name userid:(NSString *)userid{
    userid = userid == nil ? @"" : userid;
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == YES){
        return [self updateObjectUsingSQL:[NSString stringWithFormat:@"delete from %@ where name=? and userid=?", T_N_DataUpdateInfo] withArgumentsArray:@[name, userid]];
    }
    return NO;
}
```

5) T_N_DataUpdateInfo表查询

```
- (NSString *)getLastEdittimeOfName:(NSString *)name{
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == YES){
        FMResultSet *set =[self queryObjectUsingSQL:[NSString stringWithFormat:@"select * from %@ where name=?", T_N_DataUpdateInfo] withArgumentsArray:@[name]];
        NSString *lastedittime;

        //从结果集中取数据
        while ([set next]) {
            lastedittime =[set stringForColumn:@"lastedittime"];
        }
        //关闭结果集
        [set close];
        //关闭数据库
        [self closeDatabaseAfterQueryDown];
        return lastedittime;
    }
    return nil;
}
- (NSString *)getLastEdittimeOfName:(NSString *)name userid:(NSString *)userid{
    userid = [userid length] == 0 ? @"" : userid;
    BOOL flag1 = [self ifTableExists:T_N_DataUpdateInfo];
    if(flag1 == YES){
        FMResultSet *set =[self queryObjectUsingSQL:[NSString stringWithFormat:@"select * from %@ where name=? and userid=?", T_N_DataUpdateInfo] withArgumentsArray:@[name, userid]];
        NSString *lastedittime;

        //从结果集中取数据
        while ([set next]) {
            lastedittime =[set stringForColumn:@"lastedittime"];
        }
        //关闭结果集
        [set close];
        //关闭数据库
        [self closeDatabaseAfterQueryDown];
        return lastedittime;
    }
    return nil;
}
```

(3) 数据库管理单例类的使用

1) 数据库迁移，在didFinishLaunchingWithOptions执行下即可：

```
//数据库迁移处理，含数据库表的创建，暂时使用同步操作，确保首页有用到本地数据库加载的地方不会因为数据库版本未迁移而发生异常，也就是说迁移完后才makeKeyAndVisible，让首页显示出来
//    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
[[MYSQLiteManager defaultManager] migrationsLocalSQLiteIfNeeded];
//    });
```

2) 数据库表的操作

```
NSString *lastEditTime = [[MYSQLiteManager defaultManager] getLastEdittimeOfName:F_V_GaoShouZiXuan];

NSString *urlstring = [NSString stringWithFormat:@"%@%@%@", YuMing3, zixuan_gaoShou, [lastEditTime length] == 0 ? @"" : [NSString stringWithFormat:@"?lastedittime=%@", lastEditTime]];
__weak typeof(self) weakSelf = self;
[self.service requestWithURLString:[urlstring stringByAddingPercentEncodingWithAllowedCharacters:[NSCharacterSet URLQueryAllowedCharacterSet]] parameters:nil type:HttpRequestTypeGet success:^(id
    responseObject) {
    NSDictionary *dic = responseObject;
    NSArray *list = dic[@"list"];
    NSMutableArray *array = [NSMutableArray array];
    [list enumerateObjectsUsingBlock:^(NSDictionary *obj, NSUInteger idx, BOOL * _Nonnull stop) {
        GaoShouZiXuanListItem *item = [[GaoShouZiXuanListItem alloc] initWithDictionary:obj];
        [array addObject:item];
    }];

    if([list count] > 0){//取网络数据，并且更新本地数据库
        NSString *lastedittimes = [dic[@"lastedittime"] isKindOfClass:[NSNull class]] ? @"" : dic[@"lastedittime"];
        if([lastEditTime length] == 0){
            [[MYSQLiteManager defaultManager] insertIntoUpdateInfoWithName:F_V_GaoShouZiXuan lastEdittime:[lastedittimes length] == 0 ? @"" : lastedittimes];
        }else{
            [[MYSQLiteManager defaultManager] upDataUpdateInfoName:F_V_GaoShouZiXuan setLastEdittime:[lastedittimes length] == 0 ? @"" : lastedittimes];
        };

        [[MYSQLiteManager defaultManager] deleteAllTableGaoShouZiXuan];
        [array enumerateObjectsUsingBlock:^(GaoShouZiXuanListItem *obj, NSUInteger idx, BOOL * _Nonnull stop) {
            [[MYSQLiteManager defaultManager] insertIntoGaoShouZiXuanWithImageurl1:obj.imageurl1 imageurl2:obj.imageurl2 useriddes:obj.useriddes title:obj.title roomid:obj.roomid selfstocksuper_id:obj.
                selfstocksuper_id];
        }];
```