**渥瑞达北美IT培训**

**云笔记概要设计**

*版本描述*

| Owner | Date | Status | Version | Comments |
| --- | --- | --- | --- | --- |
| 张荣霞 | 2014-02-10 | Draft | 1.0 | 第一个版本 |

目录

一、系统设计 2

1.1、系统模块 2

1.2、类与类图 3

二、时序图 7

2.1、程序启动 7

2.2、登陆注册的时序图 8

2.3、添加文字笔记 9

2.4、修改文字笔记 10

2.5、删除文字笔记 12

2.6、文字笔记排序 13

2.7、文字笔记上传 13

2.8、文字笔记下载 15

2.9、添加相册笔记 16

2.10、删除相册笔记 16

2.11、上传相册笔记 18

2.12、下载相册笔记 19

三、物理视图 20

四、接口说明 21

五、ASIHttpRequest基础说明 24

5.1、下载数据 24

5.2、上传数据 25

5.3、请求队列 25

六、xml文件说明与解析 27

6.1、XML文件说明 27

6.2、XML解析 27

七、核心框架使用范例 28

7.1、网络集成简介 28

7.2、modal数据模型 28

7.3、分析网络请求类 29

7.4、数据解析类 31

7.5、引擎 34

7.6、对引擎的使用 36

八、项目风险评估 39

九、项目进度安排 39

# 一、系统设计

## 1.1、系统模块

本项目分为以下几个模块：

1.1.1、 MainViewController

在此模块中，利用Engine从和 NSNotificationCenter从网络获取数据、解析数据并将数据显示在视图中。

1.1.2、Engine

此模块，集成了TodoDb、HTTPCommunication以及XMLParser类的对象，向MainViewController提供统一接口。

1.1.3、TodoDb

TodoDb主要是对笔记列表进行统一管理，如把所有笔记保存到本地或从本地文件读取所有的笔记并进行显示等。

1.1.4、HttpCommunication

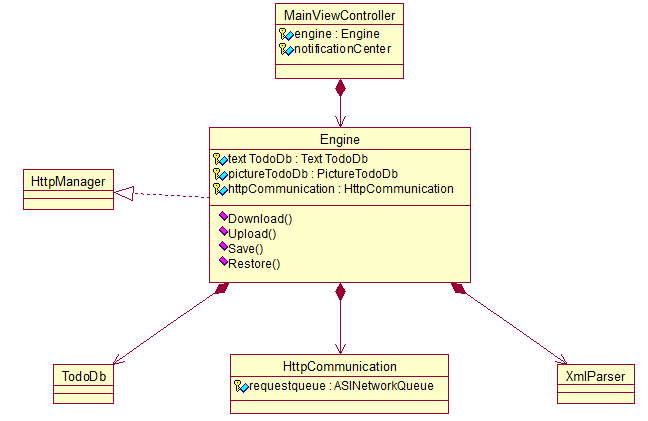
此模块封装了与网络进行交互（上传和下载）的各个方法。

1.1.5、XMLParser

此模块封装了解析网络数据的方法。

## 1.2、类与类图

1.2.1项目类图



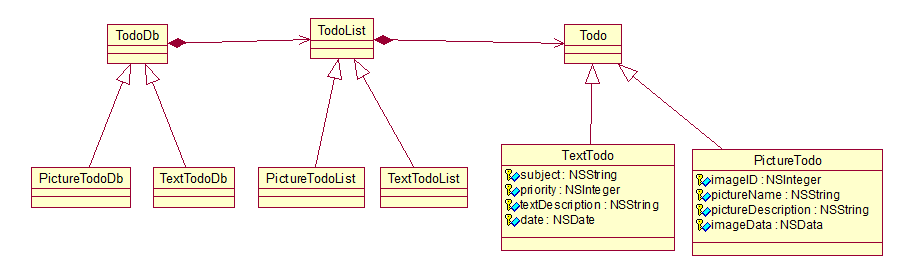
类图描述如下：

|  |  |
| --- | --- |
| ClassName | Description |
| MainViewController | 1）此类中包含一个Engine类型的实例变量engine和NSNotificationCenter的实例变量notificationCenter  2）当需要执行增删改等操作时，均通过Engine提供的相关接口实现  3）当需要执行上传操作时，通过Engine提供的上传接口执行上传  4）当需要执行下载操作时，通过Engine提供的下载接口执行下载 |
| Engine | 此类为引擎类，此类封装了网络请求类、解析类和Modal类，向UI（用户界面）提供统一接口。此类中包含了TodoDb、HttpCommunication以及XMLParser等类型的对象，其中XMLParser类型的对象可以设置为静态全局对象，并且还有Upload、Download、Restore以及Save等方法分别可以实现上传、下载、恢复以及保存数据的操作  1）当上传方法被执行时，是通过实例变量httpCommunication(HttpCommunication类型)执行上传，当某一个上传请求完成时，执行本类实现的HttpManager协议（在HttpCommunication定义）中的方法处理（解析）请求结果，请求结果由xmlparser(XMLParser类的对象)负责解析，解析完成之后，Engine会向应用程序（UIApplication）发送通知，这些通知由MainViewController负责监听，一旦监听到相应的通知，就可以弹出警告框给用户对应的提示  2）当下载方法被执行时，是通过实例变量httpCommunication(HttpCommunication类型)执行下载，当某一个下载完成时，执行本类实现的HttpManager协议（在HttpCommunication定义）中的方法处理（解析）请求结果，请求结果由xmlparser(XMLParser类的对象)负责解析，解析完成之后，Engine中会向应用程序（UIApplication）发送通知，这些通知由MainViewController负责监听，监听到之后MainViewController或其它视图控制器刷新表格显示最新下载的笔记 |
| HttpManager | 这是在引擎中声明的协议，引擎实现了此协议，所以它们之间是实现的关系，用空箭头和虚线表示。协议中可以指定请求完成所执行的方法 |
| XMLParser | 此类封装了对请求得来的网络数据进行解析的各种方法 |
| HttpCommunication | 此类封装了与网络交互的各种方法，其中包含一个requestQueue对象，可用于管理所有的请求。 |
| ToDoDb | 此类为Modal类中的其中一个类，其中封装了对Modal类的所有操作，如增加一条笔记，删除一条笔记等。 |

1.2.2、TodoDb及其子类的类图

本项目的modal数据模型，包含以下类：

ToDo，TextToDo,PictureToDo,ToDoList,TextToDoList,PictureToDoList，ToDoDB,TextToDoDB,PictureToDoDB,它们之间的关系图如下所示：



说明：

TextTodo文字笔记；

PictureTodo图片笔记；

Todo图片笔记和文字笔记的父类，在此类中可以把两者共同的属性和行为进行封装;

TextTodoList文字笔记列表；

PictureTodoList图片笔记列表；

TodoList文字笔记列表和图片笔记列表的父类，此类可以把两者的共同属性和行为封装；

TextTodoDb文字笔记列表管理类，此类中可以使用文件实现文字笔记的持久化保存（本地保存）；

PictureTodoDb同TextTodoDb;TodoListDb为TextTodoDb和PictureTodoDb的父类，可以封装两者共同的属性和行为。

下面，提供了部分子类的头文件：

@interface TextToDo : NSObject<NSCopying,NSCoding> {

NSString\* subject;//主题

NSString\* todoDescription;//描述

NSInteger priority;//优先级

NSDate\* date;//日期

}

@property(nonatomic,copy)NSString\*subject,\* todoDescription;

@property(nonatomic,copy)NSDate \*date;

@property(nonatomic,assign)NSInteger priority;

-(id)initWithToDo:(NSString\* ) \_subject

andtodoDescription:(NSString\* ) \_todoDescription

andpriority:(NSInteger)\_priority

anddate:(NSDate\*) \_date; //初始化

-(NSComparisonResult)compareSubject:(id)element;//按主题进行比较

-(NSComparisonResult)comparePriority:(id)element;//按优先级进行比较

-(NSComparisonResult)compareDate:(id)element;//按日期进行比较

@end

#import <Foundation/Foundation.h>

#import "ToDo.h"

@interface TextToDoList : NSObject<NSCoding> {

NSMutableArray\* todolistArray;

}

@property (nonatomic,retain) NSMutableArray\* todolistArray;

-(id)init;//初始化方法

-(void) addToDo:(ToDo\*) aToDo; //添加笔记

-(void)removeToDo:(NSUInteger)aIndex;//删除指定笔记

-(void)removeAll;//删除所有笔记

-(void) insertToDo:(ToDo\*) aToDo atIndex:(NSUInteger)aIndex;//在指定位置插入笔记

-(ToDo\*)todoAtIndex:(NSUInteger)aIndex;//获得指定位置的元素

-(NSUInteger)indexOfToDo:(ToDo\*)aToDo;//获得指定元素的下标

-(NSUInteger)count;//返回元素个数

-(void)sortPriority;//按优先级进行排序

-(void)sortSubject;//按主题进行排序

-(void)sortDate;//按日期进行排序

@end

#import <Foundation/Foundation.h>

#import "ToDoList.h"

@interface TextToDoDB : NSObject <NSCoding,NSCopying>{

NSString\* fileName; //文件名

ToDoList\* todoList; //todoList对象

}

@property(nonatomic,copy)NSString\* fileName;

@property(nonatomic,retain)ToDoList\* todoList;

-(id)initWithFileName:(NSString\*)aFileName;//初始化

-(void)addToDo:(ToDo\*)aToDo;//往当前列表中添加笔记

-(void)write;//把笔记列表写入文件，实现持久化保存（本地保存）

-(NSMutableArray\* )read;//读取文件中数据

-(ToDo\*)todoAtIndex:(NSUInteger)aIndex;//获得指定的笔记

-(void)insertToDo:(ToDo\*)aToDo atIndex:(NSUInteger)aIndex;//在指定位置插入笔记

-(NSUInteger)indexOfToDo:(ToDo\*)aToDo;//获得笔记的下标

-(NSUInteger)count;//获得笔记个数

-(void)removeToDo:(NSUInteger)aIndex;//移除指定笔记

-(void)removeAll;//删除所有笔记

-(void)sortPriority;//按优先级对所有笔记进行排序

-(void)sortSubject;//按主题对所有笔记进行排序

-(void)sortDate;//按日期对所有笔记进行排序

@end

#import <Foundation/Foundation.h>

@interface PicTodo : NSObject<NSCoding>

{

NSInteger imageID;

NSString\* pictureName;

NSString\* todoDescription;

NSData\* imageData;

}

@property(nonatomic,retain)NSString\* todoDescription;

@property(nonatomic,retain)NSData\* imageData;

@property(nonatomic,retain)NSString\* pictureName;

@property(nonatomic,assign)NSInteger imageID;

-(id)initWithImage:(NSData\*)newData andDescription:(NSString\*)newDescription;

@end

#import <Foundation/Foundation.h>

#import "picTodo.h"

@interface PicTodoList : NSObject<NSCoding>

{

NSMutableArray\* picTodoArray;

}

@property(nonatomic,retain)NSMutableArray\* picTodoArray;

-(id)init;

-(void) addToDo:(PicTodo\*) aToDo;

-(void)removeToDo:(NSUInteger)aIndex;

-(void)removeAll;

-(void) insertToDo:(PicTodo\*) aToDo atIndex:(NSUInteger)aIndex;

-(PicTodo\*)todoAtIndex:(NSUInteger)aIndex;

-(NSUInteger)indexOfToDo:(PicTodo\*)aToDo;

-(NSUInteger)count;

@end

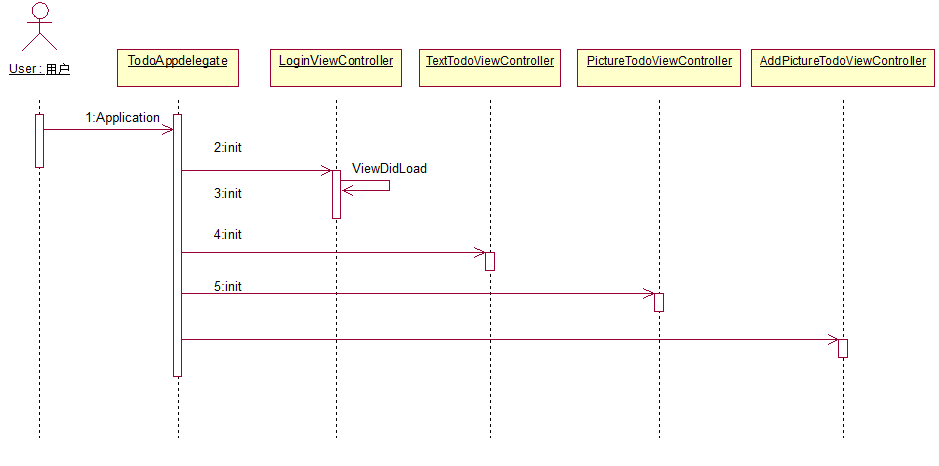
# 二、时序图

序列图主要用于展示对象之间交互的顺序。

序列图将交互关系表示为一个二维图。纵向是时间轴，时间沿竖线向下延伸。横向轴代表了在协作中各独立对象的类元角色。类元角色用生命线表示，当对象存在时，角色用一条虚线表示，当对象的过程处于激活状态时，生命线是一个双道线。

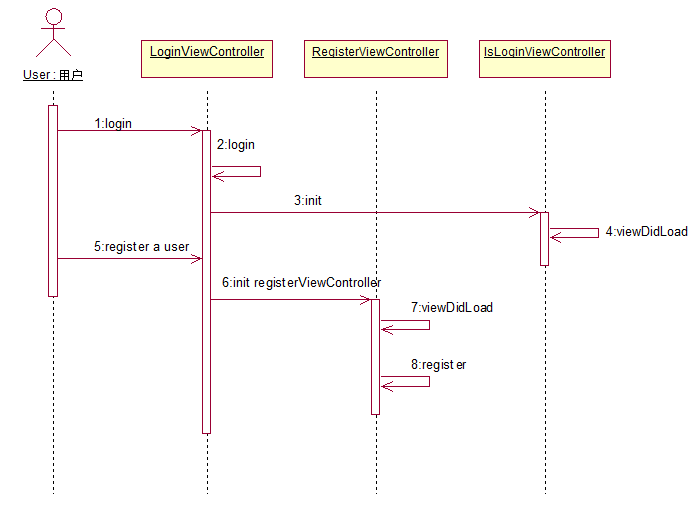
消息用从一个对象的生命线到另一个对象生命线的箭头表示。箭头以时间顺序在途中从上到下排列。以下是各个模块执行时对应的时序图

## 2.1、程序启动



当用户启动程序时，系统会创建一个Application对象并向其代理TodoAppDelegate发送相关生命周期消息。TodoAppDelegate的生命周期方法- (BOOL)application: didFinishLaunchingWithOptions:中，分别向LoginViewController、TextTodoViewController、PictureTodoViewController以及AddPictureTodoViewController发送init消息创建四个类的实例化对象。程序显示出来时，系统会向LoginViewController的对象发送viewDidLoad消息，显示登陆界面，供用户进行登陆，登陆之前，其它界面均不可见。

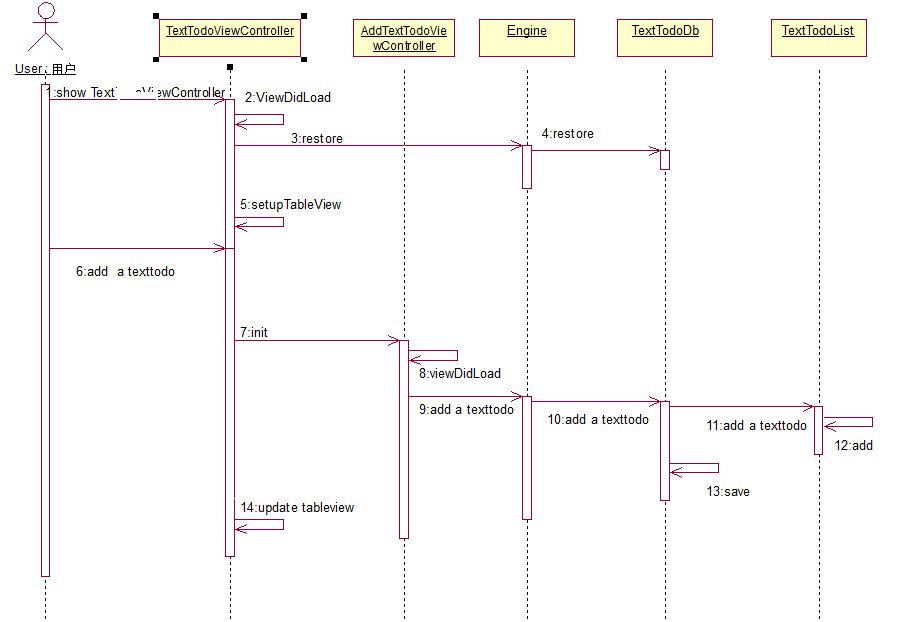
## 2.2、登陆注册的时序图



当用户登陆时，系统会向LoginViewController类自身对象发送login消息进行登陆。另外如果用户想进行注册操作，则需要初始化一个RegisterViewController对象，然后向此对象发送register消息实现注册。

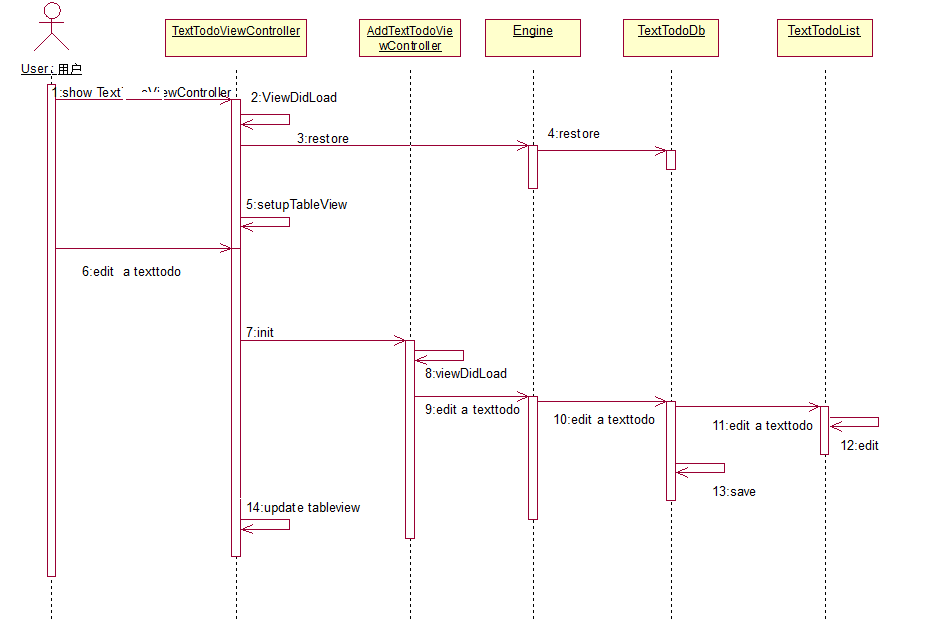
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让进行登陆 |
| 2 | 当用户名和密码指定完之后，向当前视图控制器对象发送login消息 |
| 3 | 如果登陆成功，则使用init方法创建一个IsLoginViewController对象 |
| 4 | 当IsLoginViewController视图显示时，执行viewDidLoad，此界面主要提示用户登陆成功 |
| 5 | 如果指定的用户名或者密码不对，用户可以进行注册 |
| 6 | 注册时，使用init方法初始化一个RegisterViewController对象，用于实现注册 |
| 7 | 当RegisterViewController对象视图加载到内存时，执行viewDidLoad方法 |
| 8 | 当用户名和密码等信息指定好之后，就可以实现注册了 |

## 2.3、添加文字笔记



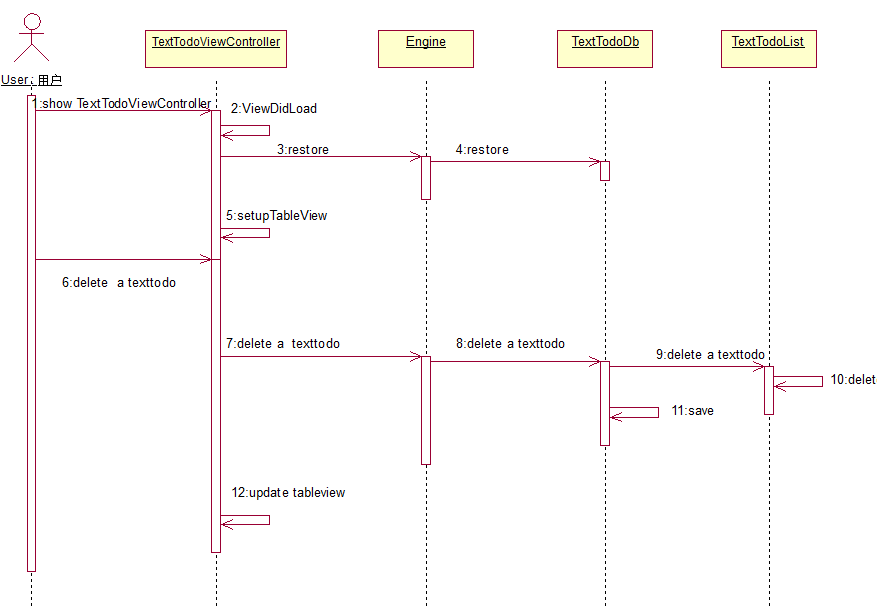
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让TextTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向TextTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要添加一条文字笔记 |
| 7 | 向AddTextTodoViewController发送init消息创建此类的实例化对象 |
| 8 | 切换到AddTextTodoViewController的视图，此时，此类对象会接收到viewDidLoad消息。 |
| 9 | 在AddTextTodoViewController的视图中设置好各个信息之后，向engine发送addTextTodo消息 |
| 10 | engine向TextTodoDb发送addTextTodo消息，以便实现添加文字笔记的操作 |
| 11 | TextTodoDb向TextTodoList发送消息，实现文字笔记的添加 |
| 12 | TextTodoList执行本类中的Add方法最终实现文字笔记的添加 |
| 13 | 成功添加完文字笔记后，TextTodoDb把新增加的文字笔记通过save方法保存到文件中 |
| 14 | 刷新表格获得最新文字笔记列表 |

## 2.4、修改文字笔记



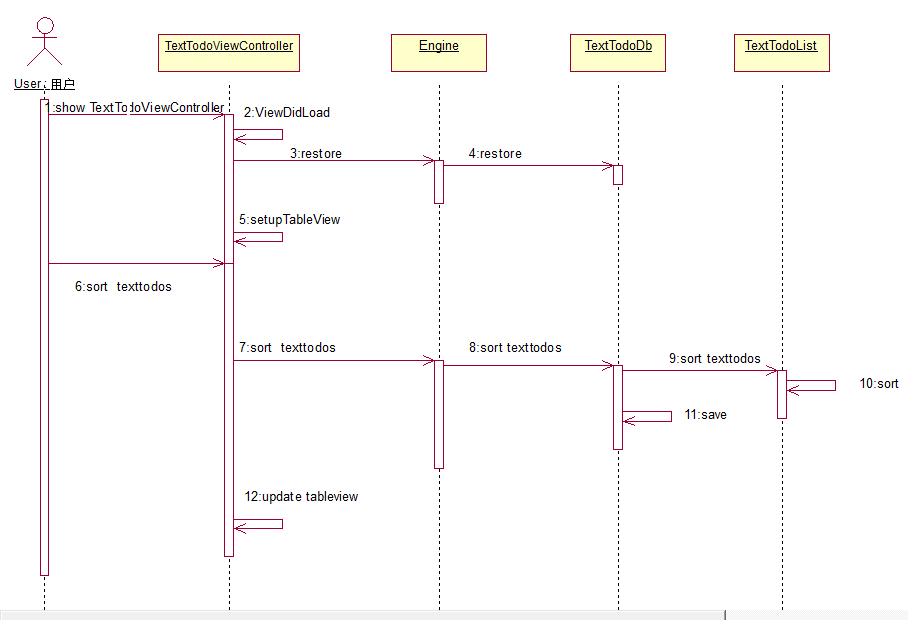
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让TextTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向TextTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要修改一条文字笔记 |
| 7 | 向AddTextTodoViewController发送init消息创建此类的实例化对象 |
| 8 | 切换到AddTextTodoViewController的视图，此时，此类对象会接收到viewDidLoad消息。 |
| 9 | 在AddTextTodoViewController的视图中设置好各个信息之后，向engine发送editTextTodo消息 |
| 10 | engine向TextTodoDb发送editTextTodo消息，以便实现修改文字笔记的操作 |
| 11 | TextTodoDb向TextTodoList发送消息，实现文字笔记的修改 |
| 12 | TextTodoList执行本类中的修改方法最终实现文字笔记的修改 |
| 13 | 成功修改文字笔记后，TextTodoDb把新增加的文字笔记通过save方法保存到文件中 |
| 14 | 刷新表格获得最新文字笔记列表 |

## 2.5、删除文字笔记



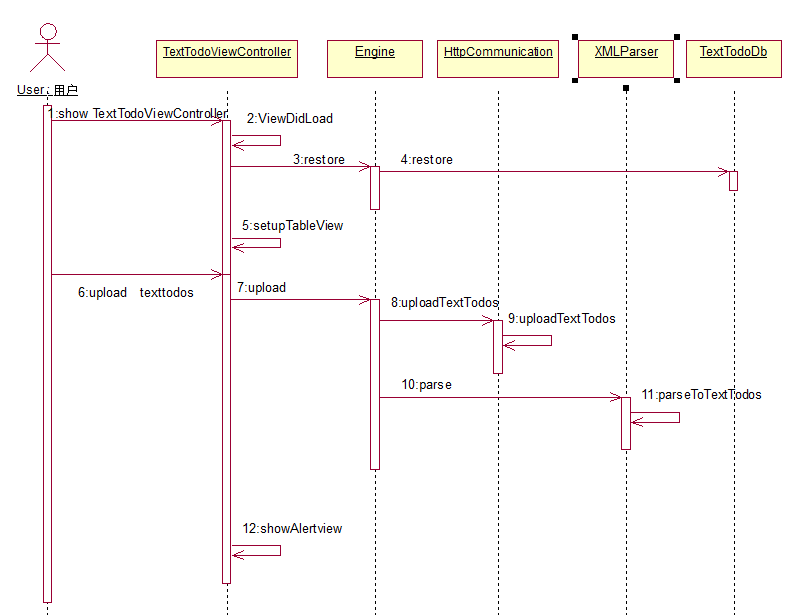
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让TextTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向TextTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要删除一条文字笔记 |
| 7 | 向engine发送deleteTextTodo消息，以便删除一条文字笔记 |
| 8 | engine向TextTodoDb发送消息，以便删除一条文字笔记 |
| 9 | TextTodoDb向TextTodoList发送删除消息，实现文字笔记的删除 |
| 10 | TextTodoList利用本类的删除方法实现删除操作 |
| 11 | TextTodoDb把最新的文字笔记列表重新保存到文件中 |
| 12 | 刷新表格，以便获得最新数据 |

## 2.6、文字笔记排序



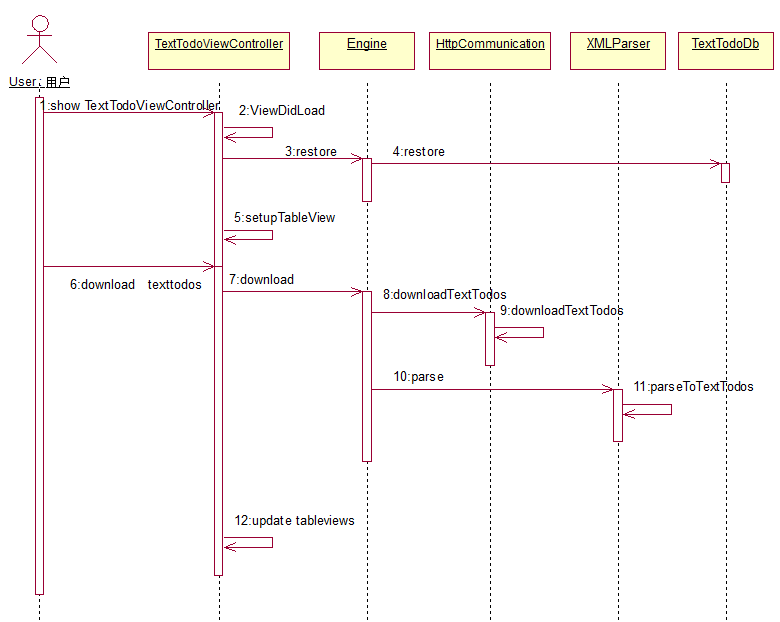
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让TextTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向TextTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要对文字笔记进行排序 |
| 7 | 向engine发送排序消息 |
| 8 | engine向TextTodoDb发送排序消息 |
| 9 | TextTodoDb向TextTodoList发送排序消息，实现文字笔记的排序 |
| 10 | TextTodoList利用本类的排序方法实现排序操作 |
| 11 | TextTodoDb把最新的文字笔记列表重新保存到文件中 |
| 12 | 刷新表格，以便获得最新数据 |

## 2.7、文字笔记上传



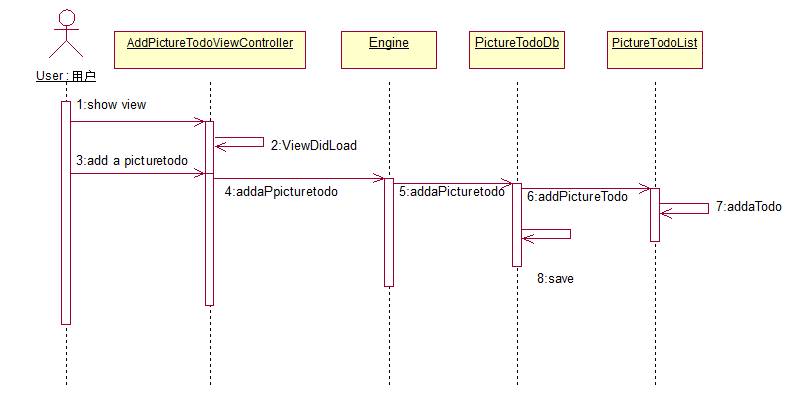
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让TextTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向TextTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要上传文字笔记 |
| 7 | 向engine发送upload消息 |
| 8 | engine向HttpCommunication发送uploadTextTodo消息 |
| 9 | HttpCommunication利用本类的upload方法实现上传操作 |
| 10 | Engine中得知上传操作结束后（可以使用代理），向XMLParser发送parse消息 |
| 11 | 在XMLParser中对网络数据进行解析 |
| 12 | 在TextTodoViewController中显示alertView，提示用户上传成功 |

## 2.8、文字笔记下载



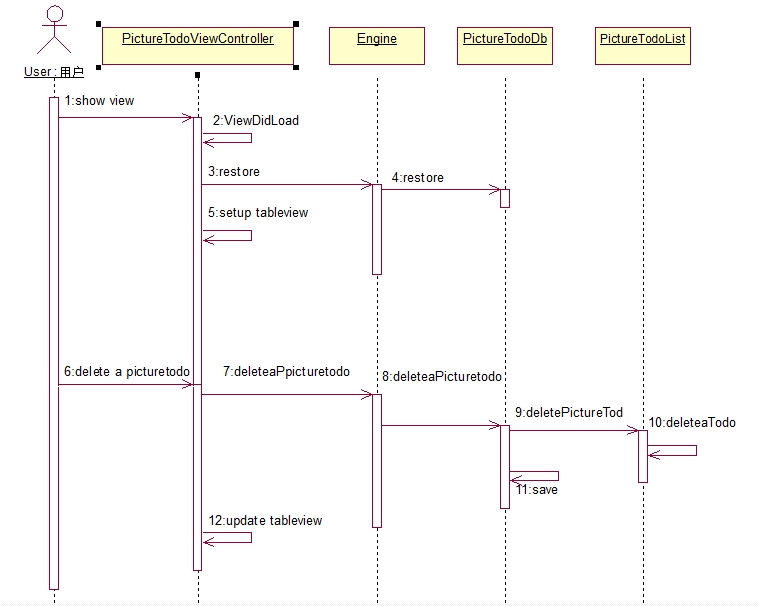
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让TextTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向TextTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要下载文字笔记 |
| 7 | 向engine发送download消息 |
| 8 | engine向HttpCommunication发送downloadTextTodo消息 |
| 9 | HttpCommunication利用本类的download方法实现下载操作 |
| 10 | Engine中得知下载操作结束后（可以使用代理），向XMLParser发送parse消息 |
| 11 | 在XMLParser中对网络数据进行解析 |
| 12 | 刷新表格显示最新下载的数据，并且可以在TextTodoViewController中显示alertView，提示用户下载成功 |

## 2.9、添加相册笔记



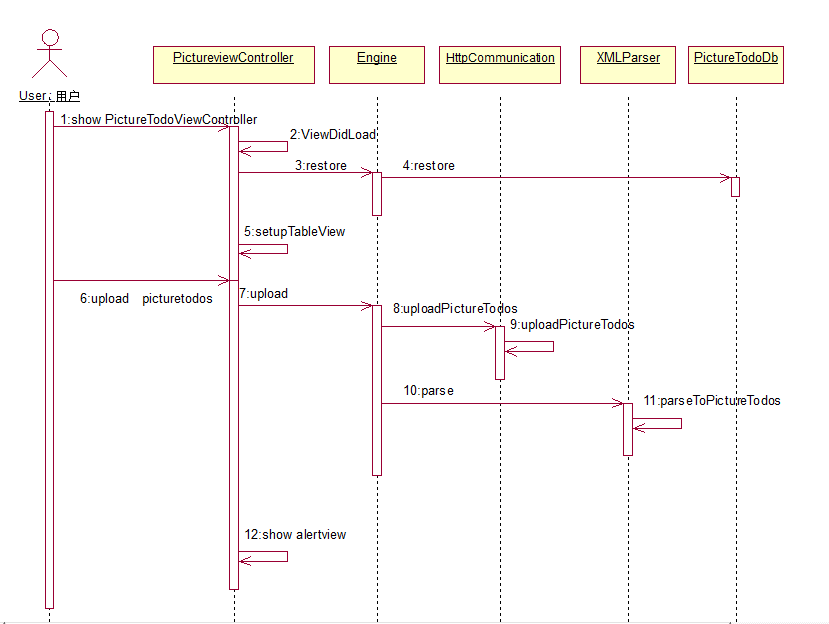
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让AddPictureTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 用户想要在此界面中实现添加一条相册笔记的功能 |
| 4 | 向engine发送addPictureTodo的消息 |
| 5 | 在Engine中向PictureTodoDb发送添加一条相册笔记的消息 |
| 6 | 在PictureTodoDb中向PictureTodoList发送添加一条相册笔记的消息 |
| 7 | 在PictureTodoList中添加一条相册笔记 |
| 8 | 在PictureTodoDb中把最新的相册笔记列表保存到本地文件中 |

## 2.10、删除相册笔记



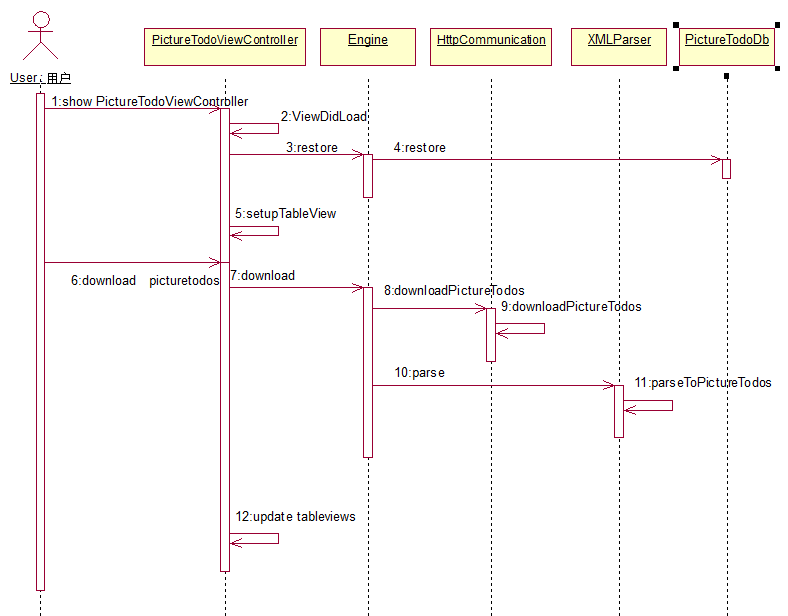
|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让PictureTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向pictureTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要删除一条相册笔记 |
| 7 | 向engine发送deletePictureTodo消息 |
| 8 | 在Engine中向PictureTodoDb发送deletePictureTodo消息 |
| 9 | 在PictureTodoDb中向PictureTodoList发送删除相册笔记的消息 |
| 10 | 在PictureTodoList中执行删除相册笔记的方法，删除指定的相册笔记 |
| 11 | 在PictureTodoDb中把最新的相册笔记列表保存到本地文件中 |
| 12 | 在PictureTodoViewController中刷新表格以获得最新数据 |

## 2.11、上传相册笔记



|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让PictureTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向PictureTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要上传相册笔记 |
| 7 | 向engine发送upload消息 |
| 8 | engine向HttpCommunication发送uploadPictureTodo消息 |
| 9 | HttpCommunication利用本类的upload方法实现上传操作 |
| 10 | Engine中得知上传操作结束后（可以使用代理），向XMLParser发送parse消息 |
| 11 | 在XMLParser中对网络数据进行解析 |
| 12 | 在PictureTodoViewController中显示alertView，提示用户上传成功 |

## 2.12、下载相册笔记



|  |  |
| --- | --- |
| Number | Description |
| 1 | 用户想要让PictureTodoViewController的视图进行显示 |
| 2 | 向当前视图控制器对象发送viewDidLoad消息把视图加载到内存中 |
| 3 | 向Engine类的对象发送restore消息，试图从本地文件中恢复数据并进行显示 |
| 4 | 在Engine中，通过向PictureTodoDb对象发送restore消息来完成数据从文件的恢复操作 |
| 5 | setupTableView并不是方法名，而是指加载表格视图的各种操作。 |
| 6 | 用户想要下载相册笔记 |
| 7 | 向engine发送download消息 |
| 8 | engine向HttpCommunication发送downloadPictureTodo消息 |
| 9 | HttpCommunication利用本类的downloadPictureTodos方法实现下载操作 |
| 10 | Engine中得知下载操作结束后（可以使用代理），向XMLParser发送parse消息 |
| 11 | 在XMLParser中对网络数据进行解析 |
| 12 | 刷新表格显示最新下载的数据，并且可以在PictureTodoViewController中显示alertView，提示用户下载成功 |

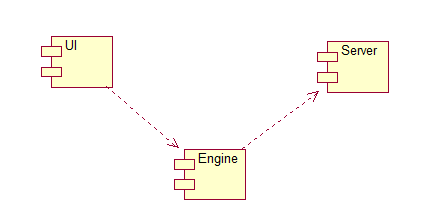
# 三、物理视图

物理视图：描述系统的非功能部分，如系统分布（软件、程序、硬件）、资源使用、可靠性以及节点和节点间的连接等。

物理视图有两种，实现视图和部署视图。

实现视图将系统中可重用的块包装成具有可替代性的物理单元，这些单元被称为构件。

本项目的实现视图如下所示

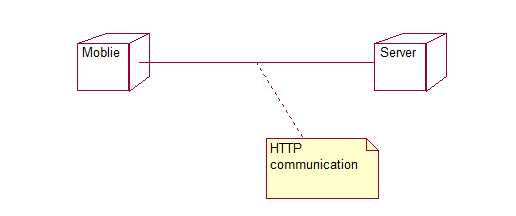


本项目UI中的数据可以通过Engine与Server进行交互。

部署视图描述位于节点实例上的运行构件实例的安排。节点是一组运行资源，如计算机、设备和存储器。这个视图允许评估分配结果和资源分配。

部署视图用部署图来表达。

本项目的部署视图如下所示



本项目的节点为手机客户端和服务器，即Moblie和Server，两者之间进行通信时，采用的是HTTP协议。

# 四、接口说明

云笔记服务器接口说明

普通笔记 客户端 服务器

|  |  |
| --- | --- |
| 接口名称 | 上传笔记XML |
| 接口URL | http://服务器网址/service/uploadusernote?**user\_name**=xxx  说明：user\_name用户名 |
| POST数据 | 笔记XML数据 |
| 服务器操作 | 接收并保存xml数据 |
| 返回值（成功） | <?xml version="1.0" encoding="utf-8"?>  <root> ／／节点：开始标签 结束标签 值  <status>1</status>  <message></message>  <errorCode>0</errorCode>  <data/>  </root> ／／存储数据 |
| 返回值（失败） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>0</status>  <message>失败信息</message>  <errorCode>0</errorCode>  <data/>  </root> |

|  |  |
| --- | --- |
| 接口名称 | 下载笔记XML |
| 接口URL | http://服务器网址/service/downloadusernote?**user\_name**=xxx  说明：user\_name用户名 |
| POST数据 | 无 |
| 服务器操作 | 读取xml数据并输出 |
| 返回值（成功） | 笔记XML数据（POST提交的xml数据，不做任何改动） |
| 返回值（失败） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>0</status>  <message>失败信息</message>  <errorCode>0</errorCode>  <data/>  </root> |

图片笔记

|  |  |
| --- | --- |
| 接口名称 | 发送图片笔记同步请求 |
| 接口URL | http://服务器网址/service/startimagenotesync?**user\_name**=xxx  说明：user\_name用户名 |
| POST数据 | 无 |
| 服务器操作 | 生成同步ID，清空原有数据 |
| 返回值（成功） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>1</status>  <message></message>  <errorCode>0</errorCode>  <data>  <sync\_id>同步ID</sync\_id>  </data>  </root> |
| 返回值（失败） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>0</status>  <message>失败信息</message>  <errorCode>0</errorCode>  <data/>  </root> |

|  |  |
| --- | --- |
| 接口名称 | 上传图片笔记 |
| 接口URL | http://服务器网址/service/uploaduserimagenote?**user\_name**=xxx  说明：user\_name用户名 |
| POST数据 | POST传递的参数：  sync\_Id 同步ID(对应的服务器中的key为sync\_id)  imageId 图片ID(对应的服务器的key为image\_id)（如果一次上传多张图片，可以用imageId区分图片，必须保证每张图片的imageId不同，并且图片的imageId必须大于0）  pictureName 图片本地文件名(对应的服务器的key为file)  todoDescription 图片说明(对应的服务器的key为image\_note)  imageData 图片数据(用户需要把此数据写入pictureName指定的文件中) |
| 服务器操作 | 接收图片数据并保存 |
| 返回值（成功） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>1</status>  <message></message>  <errorCode>0</errorCode>  <data/>  </root> |
| 返回值（失败） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>0</status>  <message>失败信息</message>  <errorCode>0</errorCode>  <data/>  </root> |

|  |  |
| --- | --- |
| 接口名称 | 下载图片笔记XML |
| 接口URL | http://服务器网址/service/downloaduserimagenote?**user\_name**=xxx  说明：user\_name用户名 |
| POST数据 | 无 |
| 服务器操作 | 将保存全部图片笔记的xml输出。 |
| 返回值（成功） | 图片笔记XML数据  <root>  <item>  <pictureNote>笔记内容1</pictureNote>  <imageName>图片url 1</imageName>  <imageId>1</imageId>  </item>  <item>  <pictureNote>笔记内容2</pictureNote>  <imageName>图片url 2</imageName>  <imageId>2</imageId>  </item>  </root>  说明: imageName为图片的URL链接地址 |
| 返回值（失败） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>0</status>  <message>失败信息</message>  <errorCode>0</errorCode>  <data/>  </root> |

注册、登录

|  |  |
| --- | --- |
| 接口名称 | 用户注册 |
| 接口URL | http://服务器网址/service/registeruser |
| POST数据 | user\_name 用户名（字母数字下划线，6-20位）  user\_password 密码（字母数字字符，6-20位） |
| 服务器操作 | 创建新用户 |
| 返回值（成功） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>1</status>  <message></message>  <errorCode>0</errorCode>  <data>  <user\_id>49</user\_id>  </data>  </root>  说明：user\_id为注册后的用户ID |
| 返回值（失败） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>0</status>  <message>失败信息</message>  <errorCode>0</errorCode>  <data/>  </root> |

|  |  |
| --- | --- |
| 接口名称 | 用户登陆 |
| 接口URL | http://服务器网址/service/userlogin |
| POST数据 | user\_name 用户名  user\_password |
| 服务器操作 | 验证用户名和密码 |
| 返回值（成功） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>1</status>  <message></message>  <errorCode>0</errorCode>  <data>  <user\_id>49</user\_id>  </data>  </root>  说明：user\_id为用户ID |
| 返回值（失败） | <?xml version="1.0" encoding="utf-8"?>  <root>  <status>0</status>  <message>失败信息</message>  <errorCode>0</errorCode>  <data/>  </root> |

# 五、ASIHttpRequest基础说明

ASIHTTPRequest 是一个直接在CFNetwork上做的开源项目，提供了一个比官方更方便更强大的HTTP网络传输的封装。它的特色功能如下：  
1、下载的数据直接保存到内存或文件系统里  
2、提供直接提交(HTTP POST)文件的API  
3、可以直接访问与修改HTTP请求与响应HEADER  
4、轻松获取上传与下载的进度信息  
5、异步请求与队列，自动管理上传与下载队列管理机  
6、认证与授权的支持  
7、Cookie  
8、请求与响应的GZIP  
9、代理请求

我们的项目中使用ASIHttpRequest第三方实现数据的与下载。

## 5.1、下载数据

下载数据可以使用ASIHTTPRequest完成

NSURL \*url = [NSURL URLWithString:@"http://www.baidu.com"];  
ASIHTTPRequest \*request = [ASIHTTPRequest requestWithURL:url];  
[request startSynchronous];//请求  
NSError \*error = [request error];  
if (!error) {//判断请求时有没有错误产生  
    NSString \*response = [request responseString];//如没有错误，则获得请求来的数据  
}

## 5.2、上传数据

上传数据，可以使用ASIFormData完成

NSURL \*url = [NSURL URLWithString:@"请求地址"];  
ASIFormDataRequest \*request=[ASIFormDataRequest requestWithURL:url];  
[request setPostValue:@"值" forKey:@"参数1"];//设置要上传的数据（键值对）  
[request setPostValue:@"值" forKey:@"参数2"];

[item setFile:@"值" forKey:@"参数3"]//如果你要上传有内容的本地文件，则可以使用setFile:forKey:完成操作，“值”就指定文件在沙盒Documents目录中的绝对路径，参数3的指定则需要根据服务器的键来指定

[request startSynchronous];//上传  
NSError \*error = [request error];  
if (!error) {//判断上传的过程中有没有错误  
NSString \*response = [request responseString];//如没有错误，则获得服务器的返回结果，根据返回结果可以得到数据上传状态（成功、失败）  
}

## 5.3、请求队列

上述例子中都只有一个请求，当有多个请求时 ,即数据获取的过程中，如果数据源复杂时，一个请求队列是必不可少的，如下所示：

- (IBAction)requestFromUrl:(id)sender {

//使用队列来管理多个请求

if (!queue) {

queue = [[ASINetworkQueue alloc] init];//创建队列

}

queue.delegate = self;//注意：为queue设置代理

ASIHTTPRequest\* request1 = [ASIHTTPRequest requestWithURL:指定的url];

[request1 setUserInfo:[NSDictionary dictionaryWithObject:值 forKey:REQUESTTYPE]];//使用usetInfo保存请求类型，以便于之后对请求进行区分

[queue addOperation:request1];//把请求添加到请求队列中

ASIHTTPRequest\* request2 = [ASIHTTPRequest requestWithURL:指定的url];

[request2 setUserInfo:[NSDictionary dictionaryWithObject:值forKey:REQUESTTYPE]];

[queue addOperation:request2];//把第二个请求加到请求队列中

ASIFormDataRequest\* request3 = [ASIFormDataRequest requestWithURL:指定的url];//用类方法创建一个用于上传数据的请求类

[request3 setUserInfo:[NSDictionary dictionaryWithObject:值forKey:REQUESTTYPE]];

[request3 setPostValue:@"值" forKey:@"user\_name"];//用于设置待上传数据，此时，必须保证key与服务器中的对应key相同，否则数据上传不成功

[request3 setPostValue:@"值" forKey:@"user\_password"];

[queue addOperation:request3];

[queue setRequestDidFinishSelector:@selector(finish:)];//当队列中的某一个请求结束时，会执行finished:方法

[queue setRequestDidFailSelector:@selector(fail:)];// 当队列中的某一个请求失败时，会执行fail:方法

[queue go];//请求队列开始执行

}

-(void)finish:(ASIHTTPRequest\*)request{

//每个请求完成时，都会执行此方法。

//获得请求的类型

RequestType type = [[[request userInfo] objectForKey:REQUESTTYPE] intValue];

if (type == 指定类型1) {//根据ASIHTTPRequest的一个属性userInfo中的信息来判断当前的请求是否是指定的请求，如果是，则可以执行你设定的操作。

//指定操作

}

else if (type ==指定类型2){

//指定操作

}

else if(type ==指定类型3){//第三个请求执行结束

//指定操作

}

}

-(void)fail:(ASIHTTPRequest\*)request{

// [[request error] localizedDescription];对错误进行处理

}

# 六、xml文件说明与解析

## 6.1、XML文件说明

XML 文档形成了一种树结构，它从“根部”开始，然后扩展到“枝叶”。

一个 XML 文档实例

XML 使用简单的具有自我描述性的语法：

<?xml version="1.0" encoding="ISO-8859-1"?>

<note>

<to>George</to>

<from>John</from>

<heading>Reminder</heading>

<body>Don't forget the meeting!</body>

</note>

第一行是 XML 声明。它定义 XML 的版本 (1.0) 和所使用的编码 (ISO-8859-1 = Latin-1/西欧字符集)。

下一行描述文档的根元素（像在说：“本文档是一个便签”）：

<note>

接下来 4 行描述根的 4 个子元素（to, from, heading 以及 body）：

<to>George</to>

<from>John</from>

<heading>Reminder</heading>

<body>Don't forget the meeting!</body>

最后一行定义根元素的结尾：

</note>

从本例可以设想，该 XML 文档包含了 John 给 George 的一张便签。

## 6.2、XML解析

解析 XML 通常有两种方式，DOM 和 SAX：

DOM解析XML时，读入整个XML文档并构建一个驻留内存的树结构（节点树），通过遍历树结构可以检索任意XML节点，读取它的属性和值。而且通常情况下，可以借助XPath，直接查询XML节点，常用的为GDataXML。

SAX解析XML，是基于事件通知的模式，一边读取XML文档一边处理，不必等整个文档加载完之后才采取操作，当在读取解析过程中遇到需要处理的对象，会发出通知对其进行处理，常用的是NSXMLParser（使用方式见课本第8章）。

GDataXML的基本使用方式如下：

读取XML文件，并解析成为DOM文档对象示例：

NSString \*filePath = [[NSBundle mainBundle] pathForResource:@"test" ofType:@"xml"];

NSData \*xmlData = [[NSMutableData alloc] initWithContentsOfFile:filePath];

NSError \*error;

GDataXMLDocument \*doc = [[GDataXMLDocument alloc] initWithData:xmlData options:0 error:&error];

if (doc == nil) {

return nil;

}

NSLog(@"%@", doc.rootElement);

[doc release];

[xmlData release];

# 七、核心框架使用范例

## 7.1、网络集成简介

网络集成是把modal、请求以及解析封装在网络引擎，由网络引擎向外提供统一接口完成与网络的交互。

## 7.2、modal数据模型

此处采用的modal为文字笔记的modal类

@interface ToDo : NSObject<NSCopying,NSCoding> {

NSString\* subject;

NSString\* todoDescription;

NSInteger priority;

NSDate\* date;

}

@property(nonatomic,copy)NSString\*subject,\* todoDescription;

@property(nonatomic,copy)NSDate \*date;

@property(nonatomic,assign)NSInteger priority;

-(id)initWithToDo:(NSString\* ) \_subject

andtodoDescription:(NSString\* ) \_todoDescription

andpriority:(NSInteger)\_priority

anddate:(NSDate\*) \_date;

-(NSComparisonResult)compareSubject:(id)element;

-(NSComparisonResult)comparePriority:(id)element;

-(NSComparisonResult)compareDate:(id)element;

@end

为了对多个todo实例 进行统一管理，增加另外一个modal类ToDoList,类的头文件声明如下：

@interface ToDoList : NSObject<NSCoding> {

NSMutableArray\* todolistArray;

}

@property (nonatomic,retain) NSMutableArray\* todolistArray;

-(id)init;

-(void) addToDo:(ToDo\*) aToDo;

-(void)removeToDo:(NSUInteger)aIndex;

-(void)removeAll;

-(void) insertToDo:(ToDo\*) aToDo atIndex:(NSUInteger)aIndex;

-(ToDo\*)todoAtIndex:(NSUInteger)aIndex;

-(NSUInteger)indexOfToDo:(ToDo\*)aToDo;

-(NSUInteger)count;

-(void)sortPriority;

-(void)sortSubject;

-(void)sortDate;

@end

这个类中是通过一个可变数组对多个文字笔记实例进行统一操作。

## 7.3、分析网络请求类

在网络请求类HttpManager中使用了ASIHTTPRequest请求数据，并且使用了ASINetworkQueue对多个请求进行统一管理，HttpManager类的声明如下：

typedef enum{

requestDone,

postDone

}RequestType;

@interface HttpManager: NSObject{

ASINetworkQueue\* requestQueue;

id<RequestDoneDelegate> delegate;

RequestType requestType;//标注请求类型以区别不同的请求

}

@property(nonatomic,retain)ASINetworkQueue\* requestQueue;

@property(nonatomic,assign)id<RequestDoneDelegate> delegate;

-(id)init;

-(void)requestFromURL:(NSURL\*)\_url;//根据给定的网址请求数据

-(void)postToURL:(NSURL\*)\_url withObject:(NSString\*)\_str;//向\_url上传\_str数据

-(void)start;//启动请求队列

@end

这个类中有一个代理，作用：当网络数据请求结束时，让代理统一对请求数据进行处理，代理在代码中的使用代码如下：

/\*当请求完成时会回调的方法\*/

-(void)requestFinished:(ASIHTTPRequest\*)request{

int requesttype = [[[request userInfo] objectForKey:@"REQUESTTYPE"] intValue];//获得请求类型

if (requesttype == requestDone) {//如果是请求指定数据，则请求类型为requestDone,可根据自己的需要设定

NSString\* responseStr = [request responseString];

[delegate requestDone:responseStr];//通过代理对请求数据进行处理

}

else if (requesttype == postDone){//如果是上传指定数据，则请求类型为postDone,可根据自己的需要设定

NSLog(@"request:%@",request);

NSString\* responseStr = [request responseString];

[delegate postDone:responseStr];//通过代理对上传之后服务器返回的字符串做处理

}

}

下面是此类的初始化方法，里面对队列进行了创建和设置：

-(id)init{

if (self = [super init]) {

requestQueue = [[ASINetworkQueue alloc] init];//创建队列

[requestQueue setDelegate:self];//设置当前类实例为队列的代理

[requestQueue setRequestDidFailSelector:@selector(requestFailed:)];//指定当队列中请求失败时回调的方法

[requestQueue setRequestDidFinishSelector:@selector(requestFinished:)];//指定当队列中的请求成功时回调的方法

}

return self;

}

如下是类中的requestFromURL:方法的实现：

-(void)requestFromURL:(NSURL\*)\_url{

//用于从\_url请求数据

ASIHTTPRequest\* request = [[ASIHTTPRequest alloc] initWithURL:\_url];// 创建请求

requestType = requestDone;

[request setUserInfo:[NSDictionary dictionaryWithObject:[NSNumber numberWithInt:requestType] forKey:@"REQUESTTYPE"]];//设置请求的Info，便于之后可以通过Info信息区分队列中的不同请求

[requestQueue addOperation:request];// 把请求放到队列中

[request release];

}

下面为向指定网址上传数据的实现：

-(void)postToURL:(NSURL\*)\_url withObject:(NSString\*)\_str{

//用于向\_url上传数据

//第二个参数为 索要上传的数据，示例为字符串

ASIFormDataRequest\* request = [[ASIFormDataRequest alloc] initWithURL:\_url];//创建请求对象

NSData\* data = [\_str dataUsingEncoding:NSUTF8StringEncoding];

NSMutableData\* data1 = [NSMutableData dataWithData:data];

[request setPostBody:data1];

requestType = postDone;//设置请求类型

[request setUserInfo:[NSDictionary dictionaryWithObject:[NSNumber numberWithInt:requestType] forKey:@"REQUESTTYPE"]];//对请求类型进行保存，以便之后方便区分各种请求

[requestQueue addOperation:request];//把请求加入请求队列

[request release];

}

下面是启动队列的方法start：

-(void)start{

[requestQueue go];// requestQueue为请求队列

}

## 7.4、数据解析类

专门负责对请求得来的数据进行解析，并且可以把解析的结果反馈给引擎类，解析类类名为NetworkParse，声明文件如下：

@interface NetworkParse : NSObject{

NSString\* parseStr;//用于接收要解析的数据

ToDoList\* datalist;//用于接收传过来的modal类对象，便于数据的存储

}

@property(nonatomic,retain)NSString\* parseStr;

@property(nonatomic,retain)ToDoList\* datalist;

-(id)initwithParseStr:(NSString\*)\_str;//初始化方法，第一个参数用于接收即将解析的数据，用于上传数据完成时对服务器返回的数据进行解析。

-(id)initwithParsrStr:(NSString\*)\_str andRecdataModal:(ToDoList\*)\_dataModal;//初始化方法，带有两个参数，第一个参数用于接收即将解析的数据，第二个参数接收盛放最终解析结果的ModaldataList

-(void)parse;//执行解析

@end

其中，两个初始化方法的实现如下：

-(id)initwithParseStr:(NSString\*)\_str{

if (self = [super init]) {

self.parseStr = \_str;

}

return self;

}

-(id)initwithParsrStr:(NSString\*)\_str andRecdataModal:(ModaldataList\*)\_dataModal{

if (self = [super init]) {

self.datalist = \_dataModal;//让实例变量和第一个参数指向同一块空间，可以实现传值的效果，方便把解析结果保存到modal中

self.parseStr = \_str;

}

return self;

}

解析方法的实现：

-(void)parse{

if (datalist) {//如果datalist不为空，则意味着这是下载数据

NSData\* data = [self.parseStr dataUsingEncoding:NSUTF8StringEncoding];

GDataXMLDocument\* document = [[GDataXMLDocument alloc] initWithData:data options:0 error:NULL];

NSLog(@"%@",document);

GDataXMLElement\* rootElement = [document rootElement];

NSLog(@"%@",rootElement);

NSArray\* array = [rootElement elementsForName:@"channel"];

NSLog(@"\*\*\*\*\*\*\*\*\*\*\*\*\*");

for (GDataXMLElement\* element in array) {

NSArray\* items = [element elementsForName:@"item"];

NSLog(@"items:%@",items);

for (int i=0; i<[items count]; i++) {

ToDo\* todo = [[ToDo alloc] init];

GDataXMLElement\* item = [items objectAtIndex:i];

todo.subject = [[[item elementsForName:@"subject"]objectAtIndex:0] stringValue];

todo.todoDescription = [[[item elementsForName:@"description"] objectAtIndex:0] stringValue];

NSString\* priorityStr = [[[item elementsForName:@"priority"]objectAtIndex:0] stringValue];

NSNumberFormatter\* numberFormatter = [[NSNumberFormatter alloc] init];

todo.priority = [[numberFormatter numberFromString:priorityStr] integerValue];

[numberFormatter release];

NSString\* dateStr = [[[item elementsForName:@"date"] objectAtIndex:0] stringValue];

NSDateFormatter\* formatter = [[NSDateFormatter alloc] init];

[formatter setDateFormat:@"yyyy-MM-dd"];

todo.date = [formatter dateFromString:dateStr];

[self.datalist addToDo:todo];

[todo release];

}

}

NSLog(@"%@",self.datalist);

}

else

{//如果datalist为空，意味着这是对上传数据时获得的服务器字符串进行解析

NSLog(@"%@",self.parseStr);

GDataXMLDocument\* docment = [[GDataXMLDocument alloc] initWithXMLString:self.parseStr options:0 error:nil];

GDataXMLElement\* rootelement = [docment rootElement];

NSLog(@"%@",rootelement);

NSString\* statusStr = [[[rootelement elementsForName:@"status"] objectAtIndex:0] stringValue];

NSLog(@"%@",statusStr);

if ([statusStr isEqualToString:@"1"]) {

NSLog(@"upload succefully");

}

}

}

## 7.5、引擎

使用引擎类Engine对数据模型类、网络请求类以及数据解析类进行封装，为外界提供统一的使用接口，引擎类的头文件如下：

//定义一个协议

@protocol RequestDoneDelegate

-(void)requestDone:(id)\_object;//当网络请求结束时会回调的方法

-(void)postDone:(id)\_object;//当上传数据结束时会回调的方法

@end

@interface Engine: NSObject<RequestDoneDelegate>{

NSString\* resultStr;//盛放请求结果字符串

ToDoList\* datalist;//用于保存解析数据

NetworkParse\* networkPaser;//用于解析

}

@property(nonatomic,retain)ToDoList\* datalist;//用于存放请求所得数据的Modal类

@property(nonatomic,retain)NetworkParse\* networkPaser;//用于对请求得来的数据进行解析的类

@property(nonatomic,retain)NSString\* resultStr;

+(NetworkIntegration\*)getInstance;//获得单实例

-(id)init;

-(void)requestFromUrl:(NSURL\*)\_requestUrl;//从执行的url中请求数据

-(void)postToUrl:(NSURL\*)\_postUrl withObject:(NSString\*)\_postStr;

@end

其中，获得本类对象的方法实现如下：

首先在实现之前需要有此声明

static NetworkIntegration\* instance = nil;

+(NetworkIntegration\*)getInstance{

if (instance == nil) {

instance = [[Engine alloc] init];//回用到本类的init方法，下面有init方法的实现

}

return instance;

}

-(id)init{

if (self = [super init]) {

resultStr = [[NSString alloc] init];

if (networkRequest == nil) {

networkRequest = [[NetworkRequest alloc] init];

}

datalist = [[ToDoList alloc] init];

}

return self;

}

下面时requestFromUrl的实现：

-(void)requestFromUrl:(NSURL\*)\_requestUrl{

networkRequest.delegate = self;//设置当前类对象为networkRequest的代理, networkRequest是HttpManager的实例

[networkRequest requestFromURL:\_requestUrl];//请求具体数据

[networkRequest start];

}

此外类中还实现了RequestDoneDelegate协议中的方法，实现如下：

-(void)requestDone:(id)\_object{//当请求结束时会回调的方法，通过代理来实现

NSLog(@"request done ,data receive by intergration");

self.resultStr = \_object;

NSLog(@"result:%@",self.resultStr);

//以下时对请求得来的数据self.resultStr进行解析

networkPaser = [[NetworkParse alloc] initwithParsrStr:self.resultStr andRecdataModal:self.datalist];//创建NetworkParse解析类的实例networkPaser对数据进行解析

[networkPaser parse];//解析

/\*解析完成之后给发送通知，便于界面的及时更新\*/

[[NSNotificationCenter defaultCenter] postNotificationName:@"PARSEFINISHED" object:nil];

}

上传数据的方法实现：

-(void)postToUrl:(NSURL\*)\_postUrl withObject:(NSString\*)\_postStr{

//用于向指定的网址上传数据,第二个参数为索要上传的数据

if (networkRequest) {

[networkRequest release];

networkRequest = [[NetworkRequest alloc] init];

networkRequest.delegate = self;

[networkRequest postToURL:\_postUrl withObject:\_postStr];

[networkRequest start];

}

}

上传结束执行的方法

-(void)postDone:(id)\_object{

//当上传数据完成时会回调的方法

NSLog(@"post done ,data receive by intergration");

self.resultStr = \_object;

NSLog(@"%@",self.resultStr);

self.networkPaser = [[NetworkParse alloc] initwithParseStr:self.resultStr];

[networkPaser parse];

//解析完成之后及时更新界面

[[NSNotificationCenter defaultCenter] postNotificationName:@"POSTDONE" object:nil];

}

## 7.6、对引擎的使用

在视图控制器类NetworkIntegrationViewController中对封装的网络引擎进行使用，头文件声明如下：

@interface NetworkIntegrationViewController : UIViewController<UITableViewDataSource,UITableViewDelegate>{

Engine \* networkIntegration;

UITableView\* tableview;//用于显示请求的数据

NSNotificationCenter\* notifCenter;//接收数据解析完成之后的消息，便于及时更新界面

NSMutableString\* postStr;//用于上传的字符串

}

@property(nonatomic,retain)NSMutableString\* postStr;

@property(nonatomic,retain)IBOutlet UITableView\* tableview;

@property(nonatomic,retain) Engine \* networkIntegration;

- (IBAction)postData:(id)sender;

- (IBAction)downloadData:(id)sender;

@end

实现文件中对网络引擎的具体使用如下：

- (void)viewDidLoad

{

[super viewDidLoad];

networkIntegration = [Engine getInstance];//获得NetworkIntegration的单一对象networkIntegration

notifCenter = [NSNotificationCenter defaultCenter]; //获得应用程序的唯一通知中心

[notifCenter addObserver:self selector:@selector(updateTable) name:@"PARSEFINISHED" object:nil]; //把当前类实例设置为 名字为:@"PARSEFINISHED"的通知的观察者，一旦程序中有:@"PARSEFINISHED"通知发出，就会被当前类实例接收并使用selectot参数中指定的方法进行处理，下面是此方法的实现，主要实现了界面的刷新

[notifCenter addObserver:self selector:@selector(uploadDone) name:@"POSTDONE" object:nil]; //把当前类实例设置为 名字为:@" POSTDONE "的通知的观察者，一旦程序中有:@"POSTDONE "通知发出，就会被当前类实例接收并使用selectot参数中指定的方法进行处理，下面是此方法的实现，主要实现了界面的刷新

//下面自己创建postStr，postStr是要上传的xml数据

}

-(void)updateTable{

[self.tableview reloadData];//界面中使用表格显示数据

}

表格中显示的行数是由networkIntegration中保存最终结果的datalist中元素个数确定，代码如下：

- (NSInteger)tableView:(UITableView \*)tableView numberOfRowsInSection:(NSInteger)section{

return [networkIntegration.datalist listcount];

}

生成单元格的方法实现代码如下：

- (UITableViewCell \*)tableView:(UITableView \*)tableView cellForRowAtIndexPath:(NSIndexPath \*)indexPath{

static NSString\* identifier = @"Cell";

UITableViewCell\* cell = [tableview dequeueReusableCellWithIdentifier:identifier];

if (cell == nil) {

cell = [[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleSubtitle reuseIdentifier:identifier] autorelease];

}

Modaldata\* tmp = [networkIntegration.datalist dataAtIndex:indexPath.row];//获得networkIntegration.datalist中下标为indexPath.row的实例tmp，tmp为Modaldata的实例

cell.textLabel.text = tmp.storeName;//用tmp的storeName给单元格的textLabel的文本赋值

cell.detailTextLabel.text = tmp.phone;// 用tmp的storeName给单元格的detailTextLabel的文本赋值

[cell.imageView setImageWithURL:tmp.pictureUrl placeholderImage:[UIImage imageNamed:@"guidex0.png"]];//此方法使用时，需要导入一第三方库SDWebImage,并包含头文件UIImageView+WebCache.h，之后才可以使用setImageWithURL: placeholderImage:方法完成网络图片的显示

return cell;

}

-(void)uploadDone{//当上传数据完成后会自行的方法，主要作用为弹出“上传完成”提示框

UIAlertView\* alert = [[UIAlertView alloc] initWithTitle:@"upload succefully" message:@"all data already upload" delegate:nil cancelButtonTitle:@"ok" otherButtonTitles:nil];

[alert show];

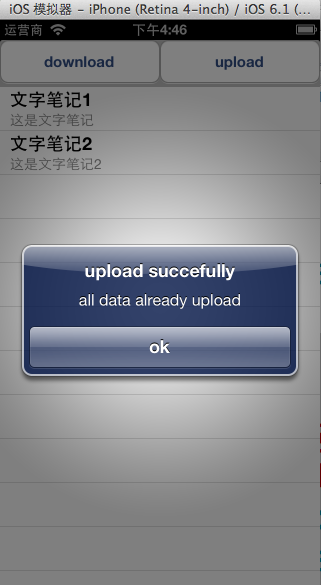
[alert release];

}

至此，所有的介绍已完成，下面是项目的运行结果图：



下载完成效果图



上传完成效果图

# 八、项目风险评估

|  |  |  |
| --- | --- | --- |
| 风险 | 概率 | 影响 |
| 可移植性差 | 60％ | 严重的 |
| 界面设计达不到预期的效果 | 30％ | 轻微的 |
| 版本更新太快 | 30％ | 轻微的 |

# 九、项目进度安排

第一天 完成Modal类

第二天 编写引擎类，实现文字笔记的增、删、改、查以及排序

第三天 完善引擎类，实现文字笔记的上传以及下载

第四天 完善引擎类，实现相册笔记的添加和删除

第五天 完善引擎类，实现相册笔记的上传和下载

第六天 添加下拉刷新等扩展操作

第七天 项目讲评