**对代码发送进行优化的即时通讯软件**

**系统设计报告**

二○二○年六月

目录

[1. 项目说明 1](#_Toc1084588394)

[1.1项目名称 1](#_Toc835942222)

[1.2项目组名 1](#_Toc842906480)

[1.3项目简介 1](#_Toc1927073748)

[1.4特殊名词解释 1](#_Toc2127602229)

[1.5目标平台 1](#_Toc860456606)

[1.6技术概要 2](#_Toc539298144)

[2. 总体设计 2](#_Toc1602915868)

[2.1客户端功能设计 2](#_Toc24641861)

[2.2系统部署结构 2](#_Toc1838897603)

[3. 概要设计 3](#_Toc1914849644)

[3.1客户端 3](#_Toc688032766)

[3.2即时通讯核心（CodeChatSDK） 4](#_Toc1714742714)

[3.3 编译服务器 7](#_Toc450251458)

[3.3.1 运行代码 7](#_Toc1791366225)

[3.3.2 同步编辑代码 7](#_Toc1918896282)

[3.4客户端界面 7](#_Toc2127884575)

[3.4.1即时通讯 7](#_Toc1310878534)

[3.4.2代码编辑 8](#_Toc900786365)

[4. 详细设计 9](#_Toc1904208852)

[4.1即时通讯核心（CodeChatSDK） 9](#_Toc89384323)

[4.1.1核心类图 9](#_Toc1191247408)

[4.1.2包图 10](#_Toc305145275)

[4.1.3关键功能实现 11](#_Toc385687889)

[4.2编译服务器 16](#_Toc1150703777)

[4.2.1 运行代码 16](#_Toc1788138804)

[4.2.2 同步编辑 16](#_Toc1362722710)

[4.3客户端界面 24](#_Toc367491715)

[4.3.1基本考虑 24](#_Toc270285233)

[4.3.2界面层级 25](#_Toc755997626)

[4.3.3关键的ViewModel 25](#_Toc1538844530)

[4.3.4关键功能实现 26](#_Toc1214454889)

[4.4 UI设计 27](#_Toc1658738335)

**对代码发送进行优化的即时通讯软件**

**系统设计报告**

1. **项目说明**

1.1项目名称

对代码发送进行优化的即时通讯软件

1.2项目组名

C#标准创造委员会

1.3项目简介

本项目的目标是开发一款即时通讯软件。其实现的基本功能有：添加好友、聊天（发送文字、表情、图片、文件等）。在此基础上，对代码的使用体验进行了优化，增加了如下功能：聊天界面代码高亮显示、利用在线编译技术可在聊天界面运行代码并显示结果、在软件内嵌的编辑器中编辑代码并运行。

1.4特殊名词解释

下面解释一些后文可能出现的、有特殊含义的名词。

（1）CAC: 本项目内部代号之一。

（2）CodeChat：本项目内部代号之一。

（3）话题：一个话题即一个聊天对话。

（4）订阅者：可理解为好友或联系人。

1.5目标平台

（1）客户端：安装Windows 10（16299及以上的版本）操作系统的x86架构计算机，需要网络连接。

（2）编译服务器：安装Linux操作系统的计算机，需要网络连接。

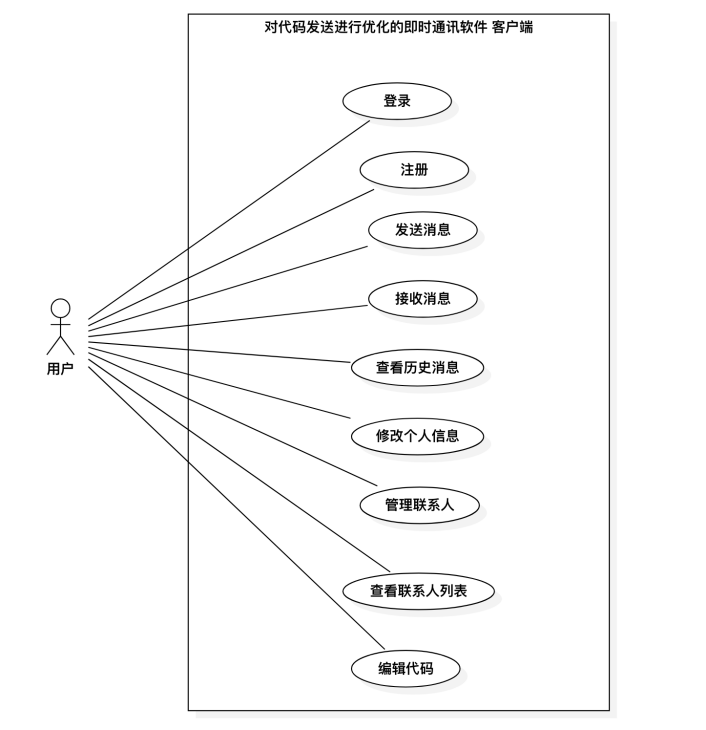
（3）即时通讯服务器：安装Linux操作系统的计算机，需要网络连接。

1.6技术概要

1. 客户端：基于UWP平台框架开发
2. 即时通讯服务器软件：tinode（开源IM服务器）
3. 编译服务器软件：基于Tomcat开发
4. 在线编译运行安全保障方案：docker容器技术
5. **总体设计**

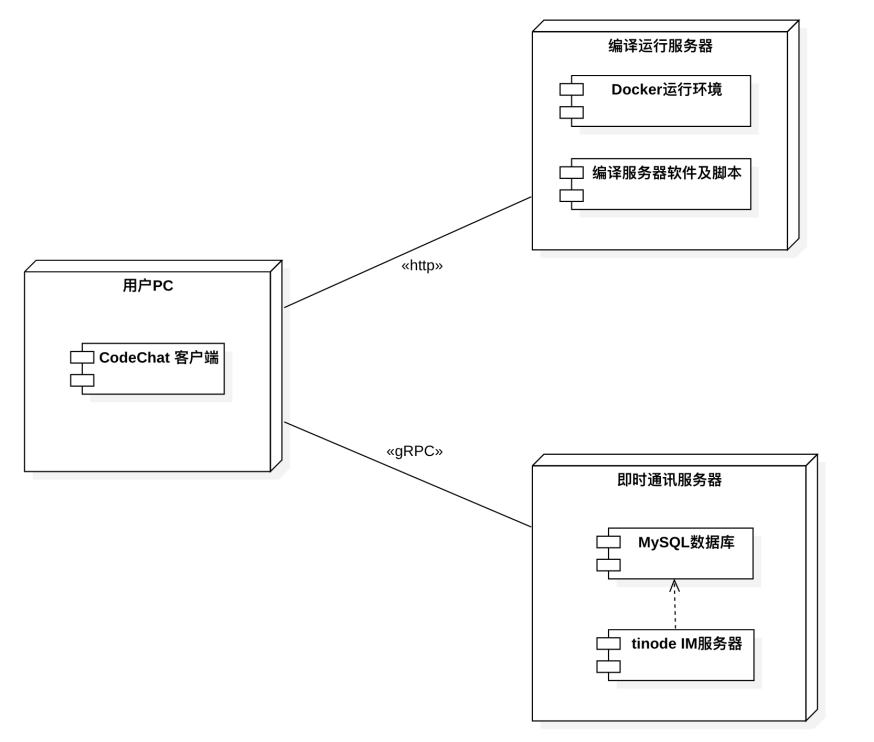
2.1客户端功能设计

客户端主要功能如下：

图**2.1.1** 客户端核心用例

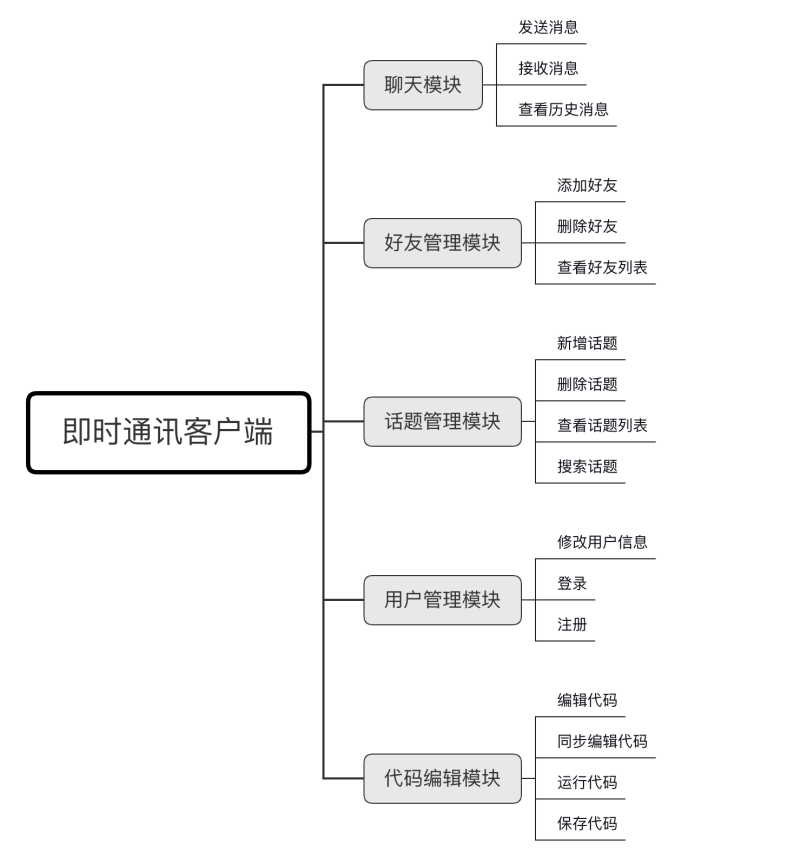
2.2系统部署结构

客户端软件安装在用户PC上。即时通讯服务器软件和编译服务器软件部署在服务器上，分别通过gRPC协议和http协议为客户端软件提供即时通讯服务和编译服务。其中即时通讯服务器采用开源服务器软件，故不在此次系统设计范围内。

图**2.2.1** 系统部署结构

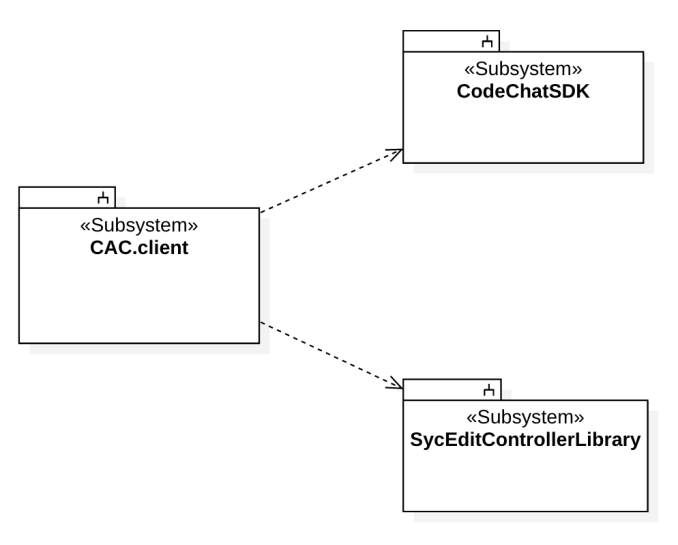
1. **概要设计**

3.1客户端

设计出客户端功能模块如下：

图**3.1.1** 客户端功能模块划分

为了方便设计开发以及代码管理，客户端分为三个工程进行开发。其中第一个工程（CAC.client）负责客户端界面部分，第二个工程（CodeChatSDK）负责客户端即时通讯核心部分，第三个工程（SycEditControllerLibrary）负责代码同步编辑部分。后两个工程不依赖第一个工程。这样的做法也降低了软件模块间的耦合，提高了可复用性。

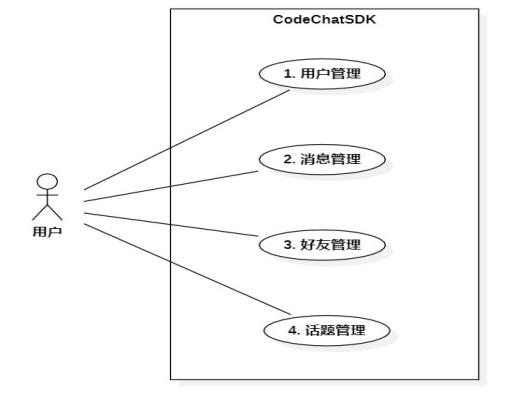
图**3.1.2** 三个工程间的依赖关系

3.2即时通讯核心（CodeChatSDK）

SDK模块主要负责对用户、消息、好友及话题进行管理，与服务器进行消息交互并负责本地数据持久化。

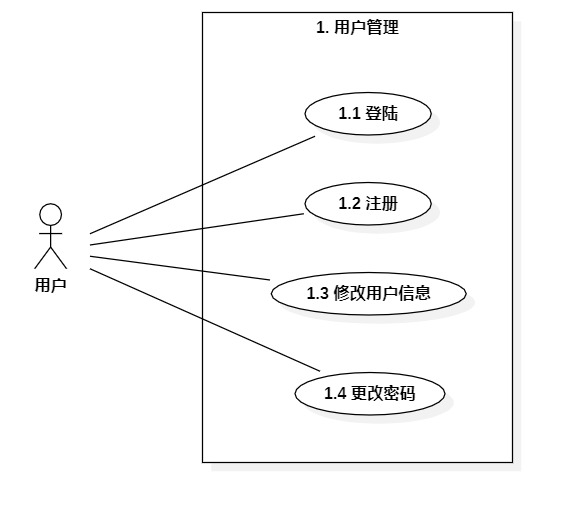
由于SDK模块涉及用例较多，故将SDK模块的用例图分为两层绘制。

顶层用例图分为四个用例，分别是用户管理、消息管理、好友管理与话题管理，顶层用例图如下所示：



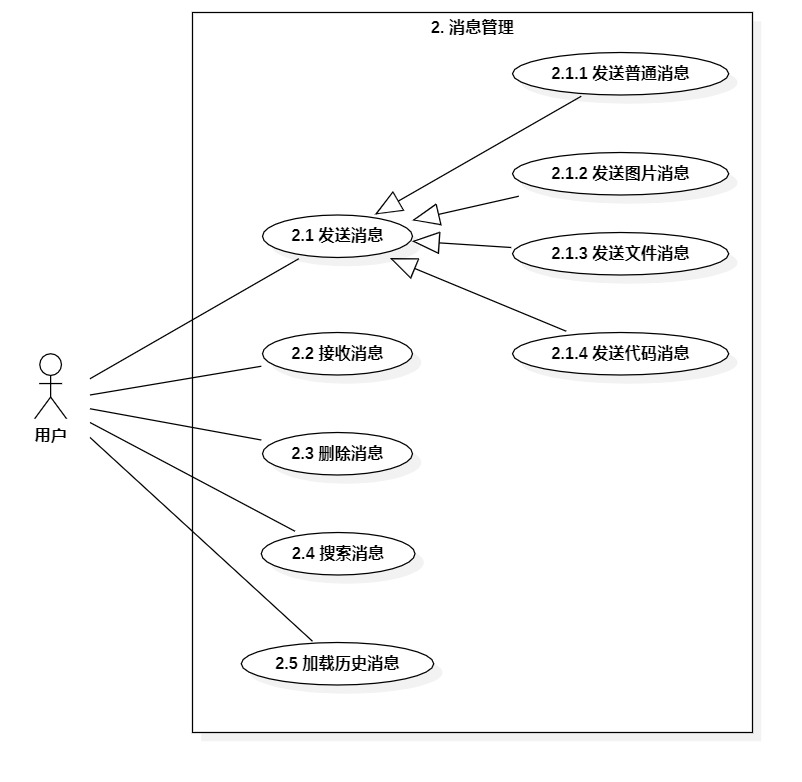
图**3.2.1**  **SDK**模块用例图

用户管理用例可细分为四个用例，分别是登陆、注册、修改用户信息与更改密码。用户管理用例图如下所示：



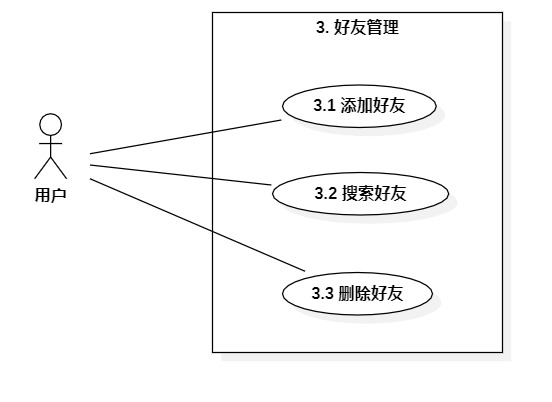
图**3.2.2** 用户管理用例图

消息管理用例可细化为五个用例，分别是发送消息、接收消息、删除消息、搜索消息与加载历史消息。其中，发送消息可泛化为发送普通消息、发送图片消息、发送文件消息与发送代码消息。消息管理用例图如下所示：



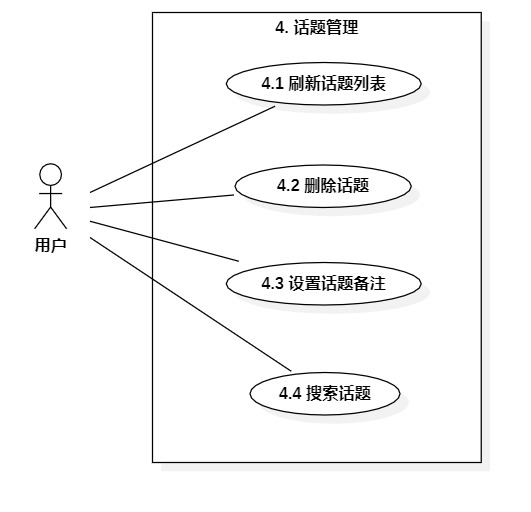
图**3.2.3** 消息管理用例图

好友管理用例可细化为三个用例，分别是添加好友、搜索好友与删除好友。好友管理用例图如下所示：



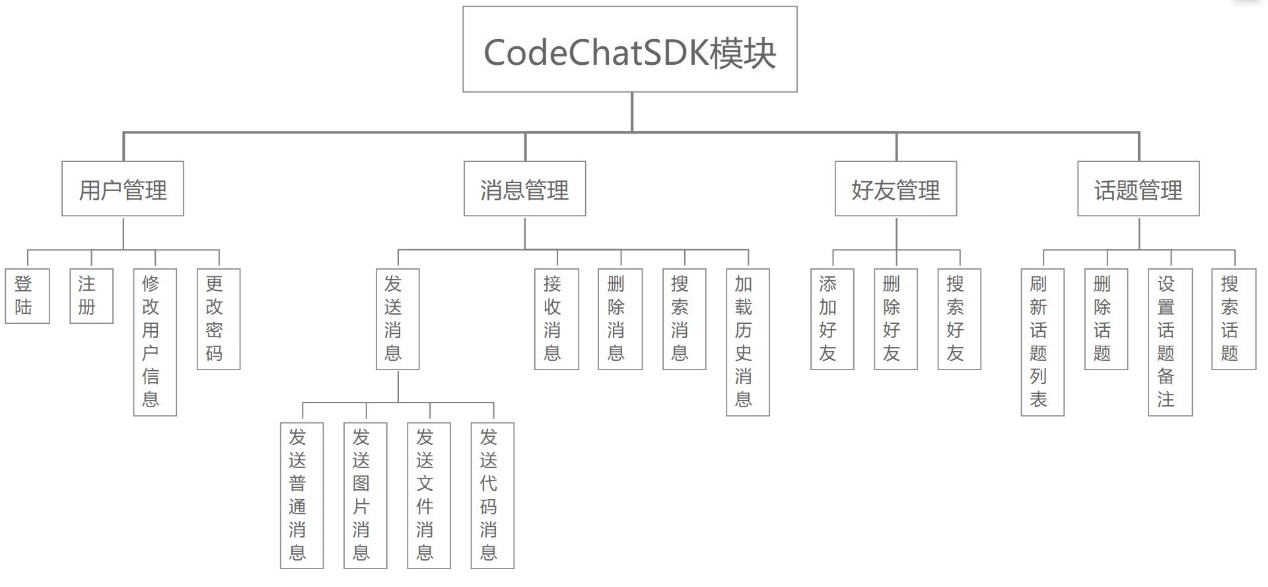
图**3.2.4** 好友管理用例图

话题管理用例可细化为四个用例，分别是刷新话题列表、删除话题、设置话题备注与搜索话题。话题管理用例图如下所示：



图**3.2.5** 话题管理用例图

根据用例图可得SDK模块的功能结构图，功能结构图如下所示：



图**3.2.6** **SDK**模块功能结构图

3.3 编译服务器

**3.3.1 运行代码**

为一款对代码优化显示的聊天软件，我们一开始就决定了要支持代码编译及运行的功能，但主要的分歧在于代码的编译和运行是在客户端上进行还是在服务器上进行，又或者是在服务器进行编译并在客户端运行。考虑到用户体验与兼容性问题，客户端的体量应当尽可能小，以及后续的软件的可扩展性问题，我们最终决定代码的编译及运行都在服务器上运行，客户端只需要发送代码到服务器即可。

**3.3.2 同步编辑代码**

同步编辑代码在网上难以找到开源的现成的与需求相匹配的库可调用，故从根本上需要由我们小组自行设计。在初期设计中，我们讨论过使用的编辑器的问题，早期设计中并没有给出详细的编辑器的相关接口的需求，尽管最后我们确定了使用开源的Monaco编辑器，但在早期设计中出于对后期扩展性的考虑以及对功能性的要求，我们决定以低耦合的模式设计同步编辑的方法库，这样可以在编辑器更换时也能很方便地实现同步编辑功能。

3.4客户端界面

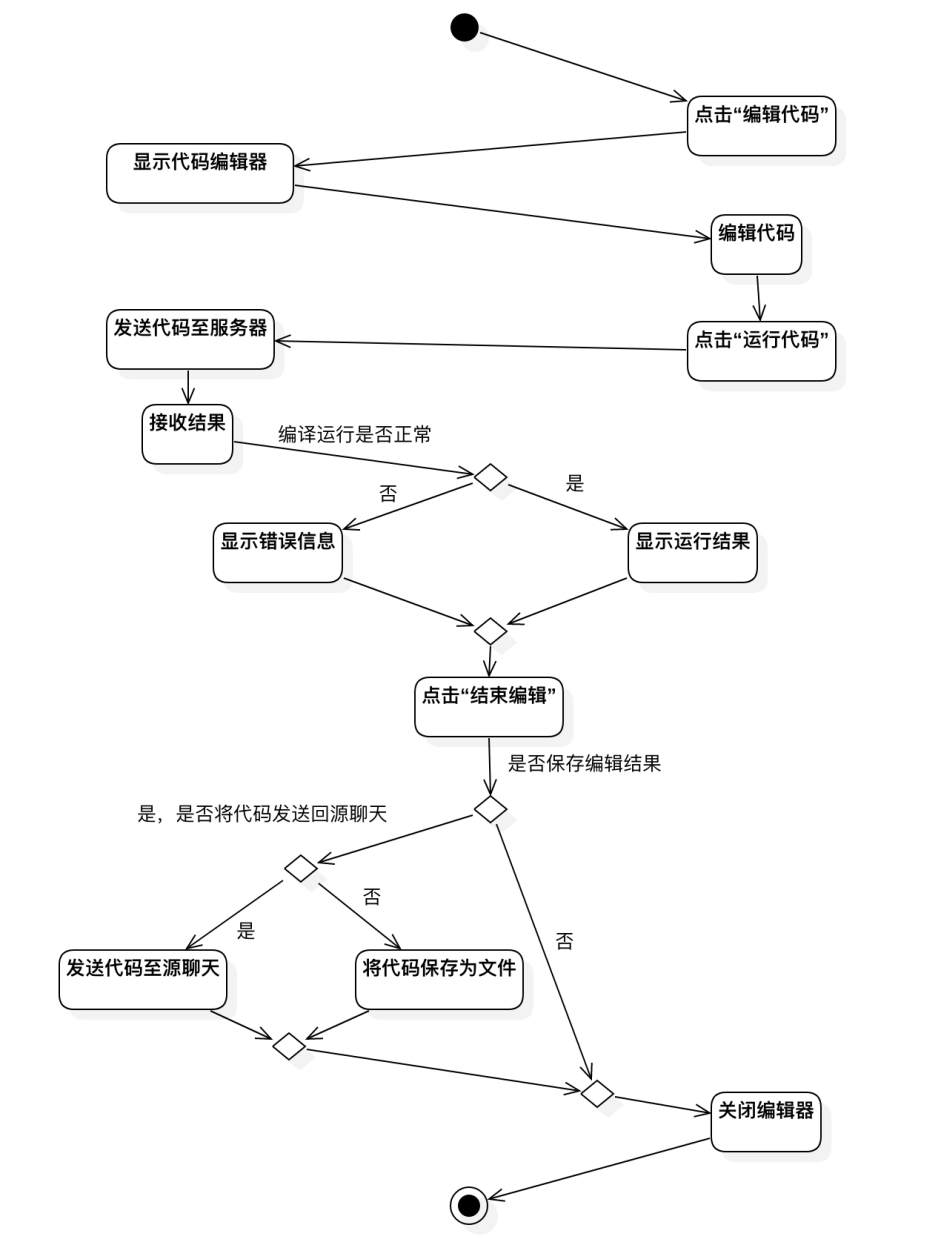
**3.4.1即时通讯**

客户端界面为即时通讯核心（CodeChatSDK）提供图形界面，故其主要功能即CodeChatSDK中的功能，上文已有描述，故不赘述。

**3.4.2代码编辑**

客户端除了具有即时通讯功能，还有代码编辑功能。由于代码较之一般文本需要更整齐的格式和关键字高亮显示等，实现此功能需要一个代码编辑器。考虑到开发一个功能完整的代码编辑器难度高、工作量大，我们决定使用开源的代码编辑器Monaco Editor。此编辑器基于Web前端技术开发，要在我们的UWP项目中运行，需要使用WebView加载。

代码编辑流程设计如下：

图**3.4.1** 代码编辑流程

1. **详细设计**

4.1即时通讯核心（CodeChatSDK）

**4.1.1核心类图**

根据用例图与需求分析，设计客户端类、实体类、控制器类及数据库访问类。

客户端类用于与服务器进行消息交互，客户端类负责进行各种请求的消息封装，并负责发送消息与接收消息，可通过修改客户端的属性实现连接不同的服务器。客户端类提供多种事件，用户可通过事件绑定进行事件处理。由于整个系统使用同一个客户端，故将客户端设计为单例。

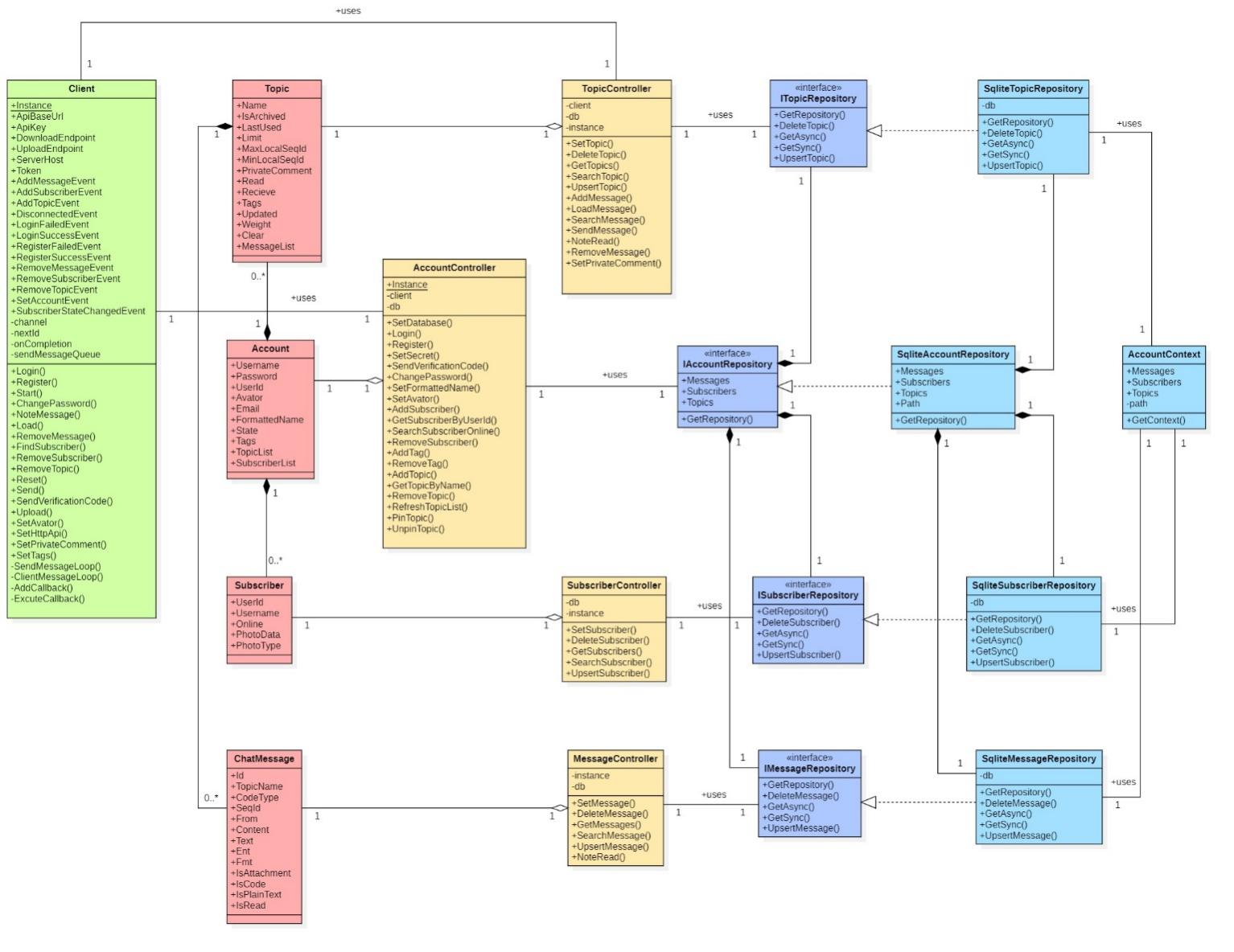
实体类主要有话题类、用户类、好友类与消息类，对应相应的实体。

控制器类主要有话题控制器、用户控制器、好友控制器与消息控制器，通过不同的控制器实现对对象的不同操作。控制器使用客户端类与数据库访问接口类，实现与服务器进行交互与本地数据的持久化。设计控制器类是为了使模块的层次更加清晰，实现业务逻辑的复用，用户可以通过调用不同的控制器实现对应的业务逻辑，而不需要重复实现业务逻辑代码。

数据库访问接口类主要有用户数据库接口、话题表接口、好友表接口及消息表接口，不同的接口定义数据操作的基本方法。设计数据库访问接口类是为了使SDK模块可以适应不同的数据库，增加SDK模块的可拓展性。

数据库访问具体实现类是对数据库访问接口的具体实现，SDK模块中已包含对SQLite数据库的实现，方便用户使用。

核心类图如下所示，其中绿色为客户端类，红色为实体类，黄色为控制器类，深蓝色为数据访问接口类，浅蓝色为数据访问具体实现类：



图**4.1.1 SDK**模块核心类图

**4.1.2包图**

根据核心类图，将SDK模块分为七个包，分别是Controllers包、Repository包、Sqlite包、Utils包、SDKClient包、Models包与EventHandler包。

Models包内是各种实体类。

EventHandler包内是各种事件类与事件参数类。

Utils包内是消息的构造器类、解析器类与转换器类，用于客户端消息的封装与服务器消息的解析。

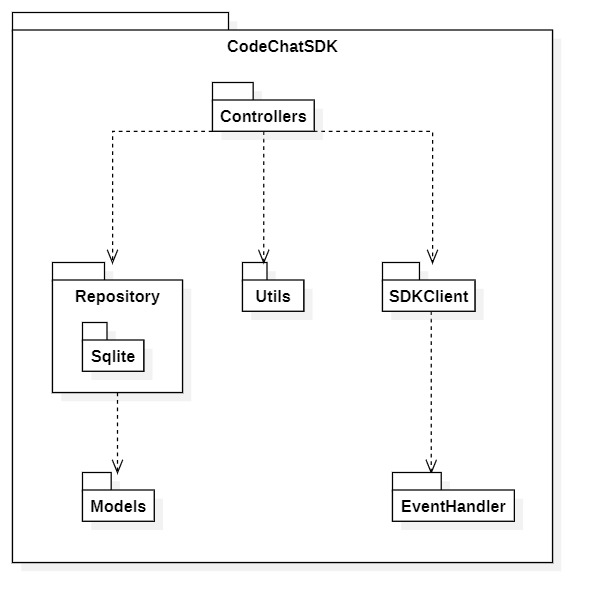
Repository包内是各种数据库访问接口类，依赖于Models包。

Sqlite包内是各种数据库访问具体实现类。

SDKClient包内是客户端类和回调类，依赖于EventHandler包。

Controllers包内是各种控制器类，依赖于Utils包、Repository包与SDKClient包。

SDK模块包图如下所示：



图**4.1.2 SDK**模块包图

**4.1.3关键功能实现**

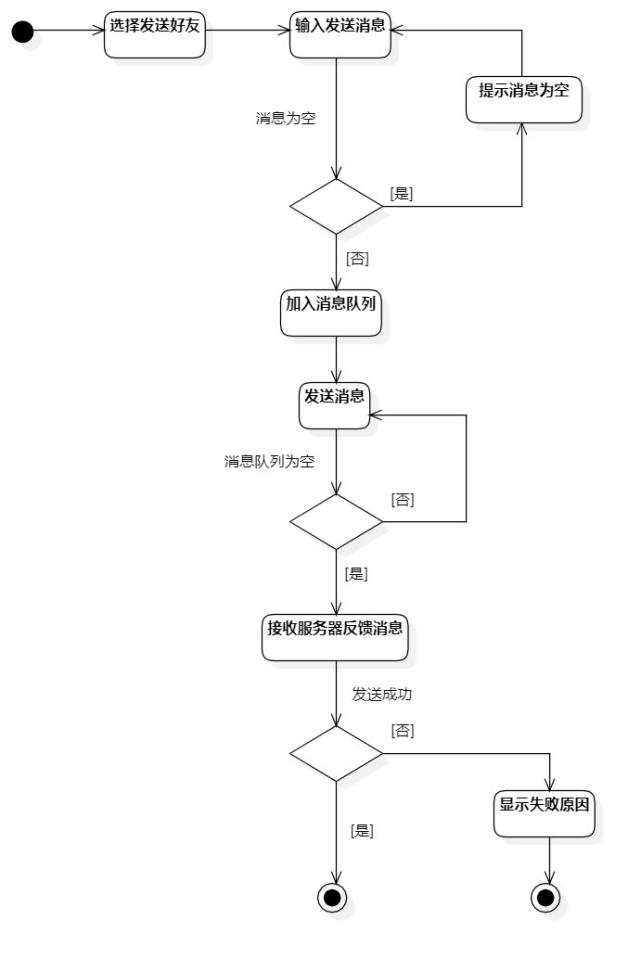
**4.1.3.1发送消息**

由于客户端与服务器之间的交互都通过消息实现，如何保证多线程并发时消息不漏发？如何确认消息已成功发送到服务器？这两个问题成为SDK模块在设计发送消息功能时必须要解决的问题。

针对以上两个问题，在客户端类中设置一个待发消息队列sendMessageQueue，用于存储待发送的消息，该队列采用ConcurrentQueue保证线程安全；同时设置回调字典onCompletion，用发送消息的ID作为键，回调对象作为值，用于等待服务器回调消息确认消息已成功发送，回调字典采用ConcurrentDictionary保证线程安全。

发送消息前需要选择发送的好友及输入消息的内容，首先判断消息是否为空，若为空则提示用户消息为空；若消息不空则将消息加入待发消息队列，若消息队列不空则持续出队，将消息发送到服务器并将回调信息加入回调字典；发送完毕后接收服务器的反馈消息，若发送不成功则提示用户发送不成功并显示原因。

发送消息的活动图如下所示：



图**4.1.3** 发送消息活动图

**4.1.3.2 加载历史消息**

为了优化软件的内存占用，故不一次性将所有消息加载到内存中。当用户需要查看较早的消息时就涉及到历史消息的加载问题。如何正确加载所需历史消息成为SDK模块在设计加载历史记录功能时必须要解决的问题。

一开始我把加载历史消息的范围确定工作交由用户处理，后来发现这样做用户将需要做多次逻辑判断才能确定范围。为了简化用户加载历史消息的流程，将加载历史消息的范围确定纳入SDK模块之中。

在用户调用加载历史消息功能时，首先检查当前消息列表是否为空，为空则说明是初次加载消息，需要从服务器请求历史消息，故将消息范围的截止序号设置为用户接收到的消息的最小序号；若不为空，则将首条消息序号作为截止序号；然后根据设定的加载条数计算出起始序号。若起始序号小于等于0则说明加载条数已越界，则将起始序号置为零。

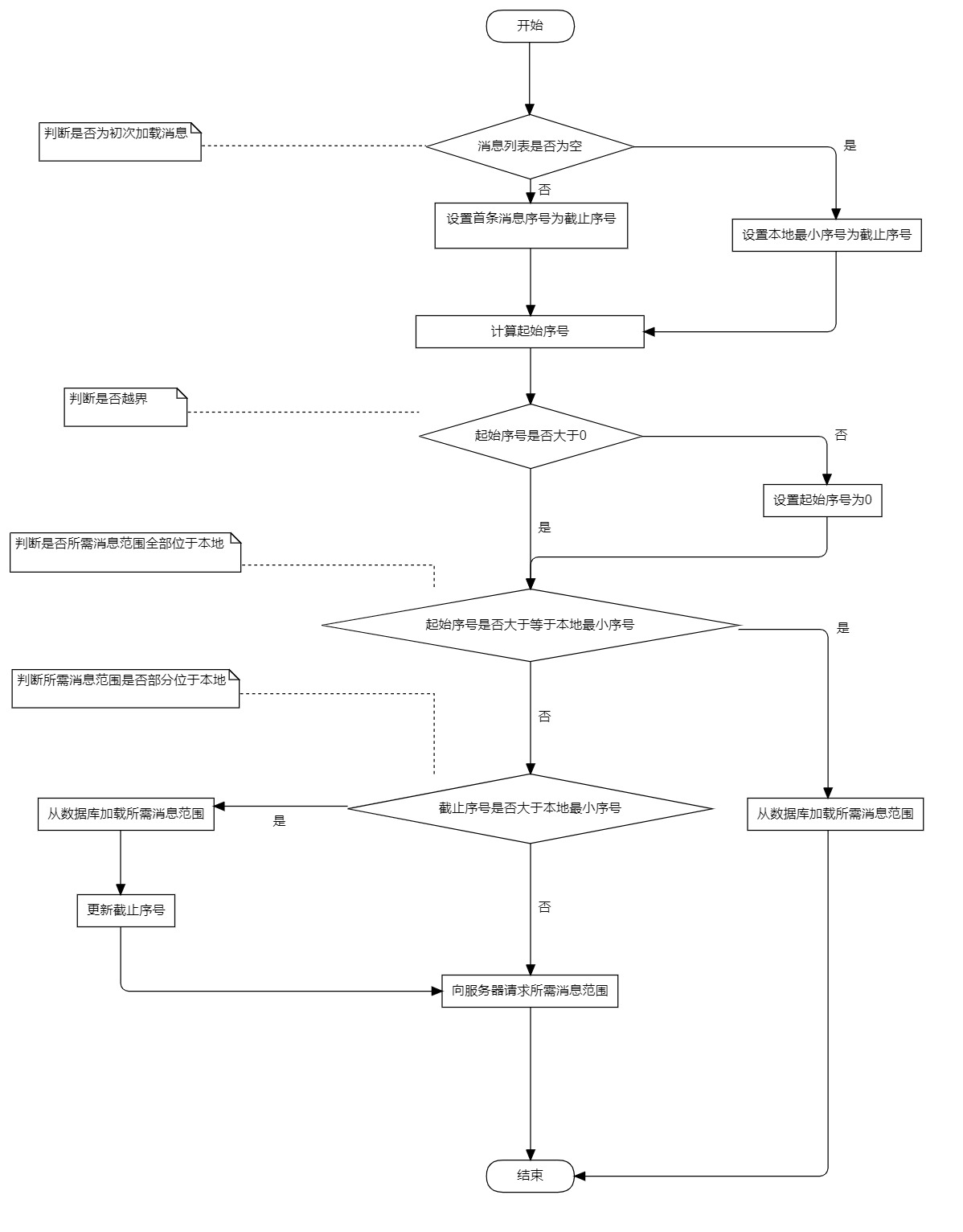
在确定起始序号与截止序号后，将起始序号与本地最小序号进行比较，若大于等于本地最小序号则说明所需消息全部位于数据库内，直接从数据库中加载即可，不需要从服务器请求历史消息；若小于本地最小序号，则继续判断截止序号是否大于本地最小序号，若大于则说明所需消息范围部分位于本地数据库，部分位于服务器，此时首先从数据库中加载所需范围的消息，更新截止序号，再向服务器请求剩余消息范围；若大于等于本地最小序号，则说明所需消息全部位于服务器，此时直接向服务器请求所需消息范围。

下面以加载条数为20，本地最小序号为12，举例说明历史消息的加载情况：

表**4.1.1** 历史消息加载说明表

|  |  |  |  |
| --- | --- | --- | --- |
| 消息列表  首条消息序号 | 起始序号 | 截止序号 | 加载情况 |
| 空 | 0 | 12 | 向服务器请求序号在[0,12)的消息 |
| 12 | 0 | 12 | 向服务器请求序号在[0,12)的消息 |
| 22 | 2 | 22 | 由数据库加载序号在[12,22)的消息；向服务器请求序号在[2,12)的消息 |
| 32 | 12 | 32 | 由数据库加载序号在[12,32)的消息 |

加载历史消息的流程图如下所示：



图**4.1.4** 加载历史消息流程图

加载历史消息代码实现如下所示：

*/// <summary>*

*/// 加载历史消息*

*/// </summary>*

**public** **async** **void** LoadMessage()

{

*//获得起始序号与截止序号*

**int** before = instance.MessageList.Count == 0 ? instance.MinLocalSeqId : instance.MessageList[0].SeqId;

**int** since = (before - instance.Limit) > 0 ? before - instance.Limit : 0;

*//判断范围*

**if** (since >= instance.MinLocalSeqId)

{

*//全部消息从数据库中加载*

MessageController messageController = new MessageController(db.Messages);

List<ChatMessage> messages = **await** messageController.GetMessages(instance, since, before) **as** List<ChatMessage>;

messages.**ForEach**(**async** m => **await** AddMessage(m));

}

**else** **if** (before > instance.MinLocalSeqId)

{

*//从数据加载部分消息*

MessageController messageController = new MessageController(db.Messages);

List<ChatMessage> messages = **await** messageController.GetMessages(instance, since, before) **as** List<ChatMessage>;

messages.**ForEach**(**async** m => **await** AddMessage(m));

*//更新截止序号*

before = instance.MinLocalSeqId;

*//剩余消息请求服务器*

client.Load(instance, since, before);

}

**else**

{

*//全部消息请求服务器*

client.Load(instance, since, before);

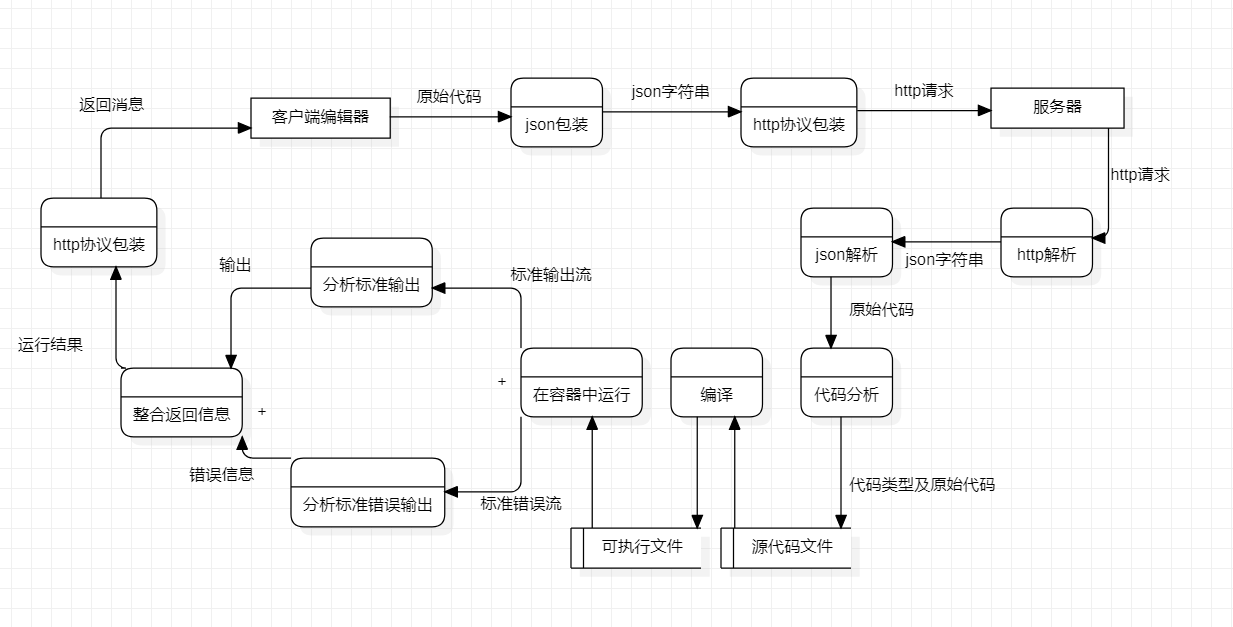
}

}

4.2编译服务器

**4.2.1 运行代码**

在编辑器中完成代码编辑后点击运行按钮会将代码及其所用语言打包成json，以解决换行符等特殊字符无法直接发送的问题，然后通过http协议将数据发送到服务器，服务器解析后根据其代码的语言类型调用适当的编译器进行编译，然后送入docker容器中运行，将运行结果返回到客户端，针对解释型语言会直接送入容器中进行解释。

图**4.2.1** 代码运行数据流图

**4.2.2 同步编辑**

为了适应各种实际具体的编辑器，我们决定将同步编辑的控制部分从编辑器里面独立出来，单独做了一个同步编辑控制的类库，并对外暴露一个接口——控制器连接器，对于具体的某个编辑器，只需要针对其特点实现该接口并连接编辑器，在控制器中调用编辑器的方法并响应编辑器的事件来完成同步的流程。

具体的同步的实现逻辑是，对于编辑器中的文本将其按行抽象为控制器中的文本行，并给每个抽象文本行一个独立的hash值作为标识，在同步过程中将按照hash值的标识来确定是对哪一个行进行了操作，以在最小的数据交流过程中保持绝对的同步。我们首先考虑了两种同步的操作的应用方式，第一种是对于同一段文本，无论同步操作的顺序如何，经过运算后都会得到相同的文本，第二种是对于同一段文本，无论双方各进行了哪些操作，只要对方的操作按照其产生的顺序到达自己的控制器，即可在双方得到相同的文本。鉴于性能和实际应用场景考虑，我们还是采用了第二种方案。

我们采用这种同步方式的初衷就是为了加强性能，减少同步时需要传递的消息，并且支持双方异步的操作能够在对方的编辑器中体现，使得对长文本的编辑也能在短时间内得到同步，于是我们将编辑器的操作抽象成了三类，第一类是在某行的后面增加新的行，第二类是删除某行，第三类是修改某行的内容，其余的文本操作均可以通过分解为这三类操作实现。为每一行设置脏位和新增行队列以标识需要同步操作的行，在接收到反馈信号后才会视情况修改脏位。行的标记有四种，在控制器中多了一种“不可修改”用于显示修改冲突行的行中对方的文本，在后面解释，这四种标记分别为“未被修改”、“已修改”、“已删除”和“行链表头”。出于性能考虑，我们并不在每次操作后向对方发送消息，而是将消息置入消息队列。双方各自维护了两个消息队列，分别存储对方向己方发送的消息和己方向对方发送的消息，在同步周期末到来时（同步周期时长默认为1s），己方向对方发送的消息队列会被发送并清空，控制器会得到对方向己方发送的消息队列。

对于修改行的操作：如果某行的标记为未修改或已被修改，在修改该行后，该行的标记均会被置为已被修改；如果某行的标记为已删除，则代表该行已在编辑器中被己方删除，无法且无需对其操作，该行无法被修改。修改发生后消息不会立即创建，而是在同步周期末，消息队列将被发送时才会根据脏位创建，由此减少不必要的操作；在收到修改消息后，会找到自己的hash值相同的对应的行，如果改行的标记为未被修改则执行修改并返回反馈信息，反馈消息中展示了被修改后行的文本，将其添加到消息队列；如果该行的标记位为被删除则不执行任何操作且不反馈消息；如果该行的标记位为被修改则发生冲突，修改消息中的信息会被显式展示在该行的后面（如何处理视具体编辑器而定，推荐采用行后异色显示，因为这与控制器中的结构逻辑相吻合），在行链表中标记为“不可修改”，不反馈消息；收到反馈消息后，如果反馈消息中的细节参数的值与行的文本相同，则将脏位修改为未被更改，否则意味着该行再次被修改，忽略此次反馈信息。

对于删除行的操作：无论该行脏位标记为何，被要求删除后均会被标记为“已删除”，并且在编辑器的实际文本中显式删除；收到删除消息后，会直接删除该行并将编辑器文本中实际地对应行删除，反馈信息；收到反馈信息后，真正移除该行，该操作的目的在于应对被删除的行可能在对方的文本中进行过增添行的操作的情况。每次删除行后消息立即被创建并置入队列。

对于增添行的操作（增添行单纯指遇到回车符产生新行，新行的文本设置应在修改行操作中完成，新增行的行文本为空值）：新增行操作指在某一行后新增一个行，操作对象为旧行，操作后产生新行。在第一行前新增行应对链表首行进行操作，链表首行不可视，它的唯一作用就是新增可视的首行；不设脏位，但会在抽象的文本行中维护一个增添行的队列，先进先出，储存新增的行的hash值；在新增行后，源操作行的队列中添加一个新值；在收到新增行的消息后，会检查源操作行的队列，如果队列为空则产生反馈消息，如果队列不为空则对队列执行dequeue操作，队列的元素减少一，将获取到的行与新行的hash值进行比较，如果新行hash值小，则将新行置于获取到的行的前面，如果新行hash值大，则将新行递归置于获取到的行的后面，此处的递归指如果dequeue操作获取到的行的队列中有值，而该值对应行的队列中无值，则将新行置于该队列首的行后，如果该值对应行的队列中有值，则继续检查，直至找到这个递归队列的队首（即文本中dequeue产生的行以及对其执行增添行操作产生的全部子行的末尾），将新行置于其后，这是保证双方最终文本完全相同的关键，有dequeue操作后不反馈消息；如果收到对方的新增行反馈，则对对应行的队列执行dequeue操作；新行每次新增行后消息立即被创建并置入队列。由于同步的双方消息机制是对称的，故剩下的情况不需要设计，脏位和队列会自动消除。以下是一个包含了几种情况的同步的实例展示。

假设原始的抽象文本为（注意标记均为U）

;NewLines

#include <iostream> ;ID:in01; Mark:U

;NewLines

using namespace std; ;ID:us02; Mark:U

;NewLines

int main() ;ID:in03; Mark:U

;NewLines

{ ;ID:us04; Mark:U

;NewLines

cout << "Hello, World!"; ;ID:co05; Mark:U

;NewLines

return 0; ;ID:re06; Mark:U

;NewLines

} ;ID:co07; Mark:U

;NewLines

此处为了清晰的展示，每两行表示实际的一行，第一行分别为hash，标记位（U：未修改，C：修改，D：删除），第二行NewLines展示的是新增队列。

我们对其作出两种不同的修改：

第一种修改：

;NewLines

#include <iostream> ;ID:in01; Mark:U

;NewLines:new01,new03

;ID:new03 Mark:U

NewLines

#include<vector> ;ID new01; Mark:C

;NewLines:new02

#include< ;ID new02; Mark:C

;NewLines:

using namespace std; ;ID:us02; Mark:U

;NewLines

int main() ;ID:in03; Mark:U

;NewLines

{ ;ID:us04; Mark:U

;NewLines

cout << "Hello, World!"; ;ID:co05; Mark:U

;NewLines

return 0; ;ID:re06; Mark:U

;NewLines

} ;ID:co07; Mark:U

;NewLines

对应的操作队列（为展示清晰修改均按顺序写出，ADD为新增，UPD为修改，DEL为删除）：

ADD in01 new01

UPD new01 #include<vector>

ADD new01 new02

UPD new02 #include<

ADD in01 new03

第二种修改：

;NewLines:te01

#include<iostream> ;ID:te01; Mark:C

;NewLines

【删除】#include <iostream> ;ID:in01; Mark:D

;NewLines:te00

# ;ID:te00 Mark:C

;NewLines:tn01,xos1

;ID:xos1 Mark:U

;NewLines

;ID:tn01 Mark:U

;NewLines:dm20

;ID:dm20 Mark:U

;NewLines

using namespace std; ;ID:us02; Mark:U

;NewLines

int main() ;ID:in03; Mark:U

;NewLines

{ ;ID:us04; Mark:U

;NewLines

cout << "Hello, World!"; ;ID:co05; Mark:U

;NewLines

return 0; ;ID:re06; Mark:U

;NewLines

} ;ID:co07; Mark:U

;NewLines

对应的操作队列：

ADD in01 te00

DEL in01

ADD head te01

UPD te00 #

UPD te01 #include<iostream>

ADD te00 tn01

ADD te00 xos1

ADD tn01 dm20

两者交换逻辑进行同步测试，最终结果为：

第一种修改同步之后：

;NewLines

#include<iostream> ;ID:te01 Mark:U

;NewLines

;ID:new03 Mark:U

;NewLines

#include<vector> ;ID new01; Mark:U

;NewLines

#include< ;ID new02; Mark:U

;NewLines:

# ;ID te00; Mark:U

;NewLines

;ID xos1 Mark:U

;NewLines

;ID tn01 Mark:U

;NewLines

;ID dm20 Mark:U

;NewLines

using namespace std; ;ID:us02; Mark:U

;NewLines

int main() ;ID:in03; Mark:U

;NewLines

{ ;ID:us04; Mark:U

;NewLines

cout << "Hello, World!"; ;ID:co05; Mark:U

;NewLines

return 0; ;ID:re06; Mark:U

;NewLines

} ;ID:co07; Mark:U

;NewLines

第二种修改同步后：

;NewLines

#include<iostream> ;ID:te01 Mark:U

;NewLines

;ID:new03 Mark:U

;NewLines

#include<vector> ;ID new01; Mark:U

;NewLines

#include< ;ID new02; Mark:U

;NewLines

# ;ID te00; Mark:U

;NewLines

;ID xos1 Mark:U

;NewLines

;ID tn01 Mark:U

;NewLines

;ID dm20 Mark:U

;NewLines

using namespace std; ;ID:us02; Mark:U

;NewLines

int main() ;ID:in03; Mark:U

;NewLines

{ ;ID:us04; Mark:U

;NewLines

cout << "Hello, World!"; ;ID:co05; Mark:U

;NewLines

return 0; ;ID:re06; Mark:U

;NewLines

} ;ID:co07; Mark:U

;NewLines

测试结果：可以看到，标记位均为U，并且新增行队列均为空，最关键的是两段文本内容，行顺序和对应的行的hash值也相同。这可以证明该算法在双方以流形式异步传输的同步session中可以使用。

下面是客户端中某一方的控制器的具体实现类图。

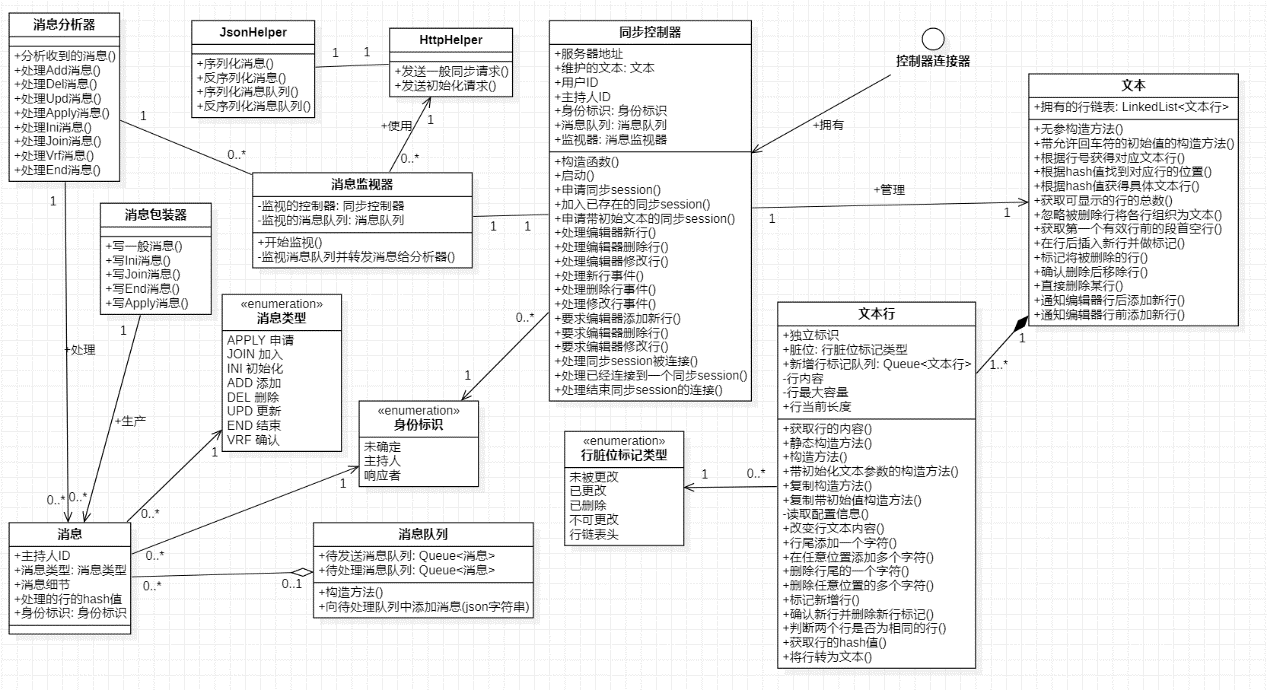
核心控制器模块包含包装器、分析器两个静态控制类和监视器、维护队列和同步控制器三个动态控制类。包装器负责将操作封装为消息，分析器用于解析消息并执行相关操作，监视器用于监控消息维护队列，以调用包装器和分析器处理业务逻辑，同步控制器为连接器接口维护的核心对象，用于执行控制操作。

Helper模块包含httpHelper和jsonHelper，负责打包封装消息并转发。

抽象数据模块有文本类、文本行类，维护了抽象的文本并提供了相关方法来操作抽象的数据。

实体类有消息类，用于将操作序列化。

控制器连接器为编辑器的控制器接口，提供了简单的必要的方法用于将同步功能初步落实到编辑器的文本中。

图**4.2.2** 同步编辑控制类图

4.3客户端界面

**4.3.1基本考虑**

客户端用户界面部分主要采用MVVM的设计思想。对于不便使用MVVM之处，使用类似MVP或MVC的设计。

界面上显示的数据在代码中使用ViewModel类进行表示。

客户端界面工程引用CodeChatSDK来对接即时通讯功能。故客户端界面类中不存在任何即时通讯实现部分的代码。这样做有效降低了耦合，但产生了一个问题，就是CodeChatSDK提供的数据类（Model）不一定符合界面的设计要求，故需要进行一层由Model到ViewModel的转换。项目中此转换由ModelConverter类的静态方法完成。

**4.3.2界面层级**

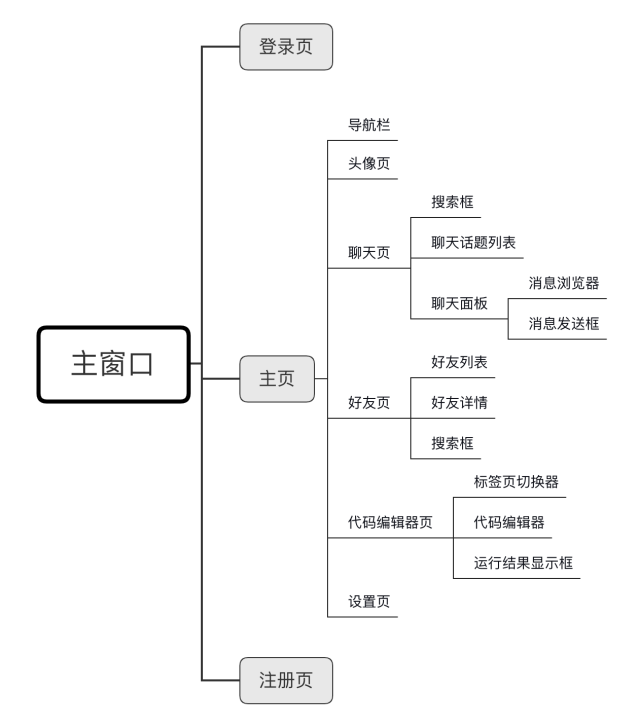
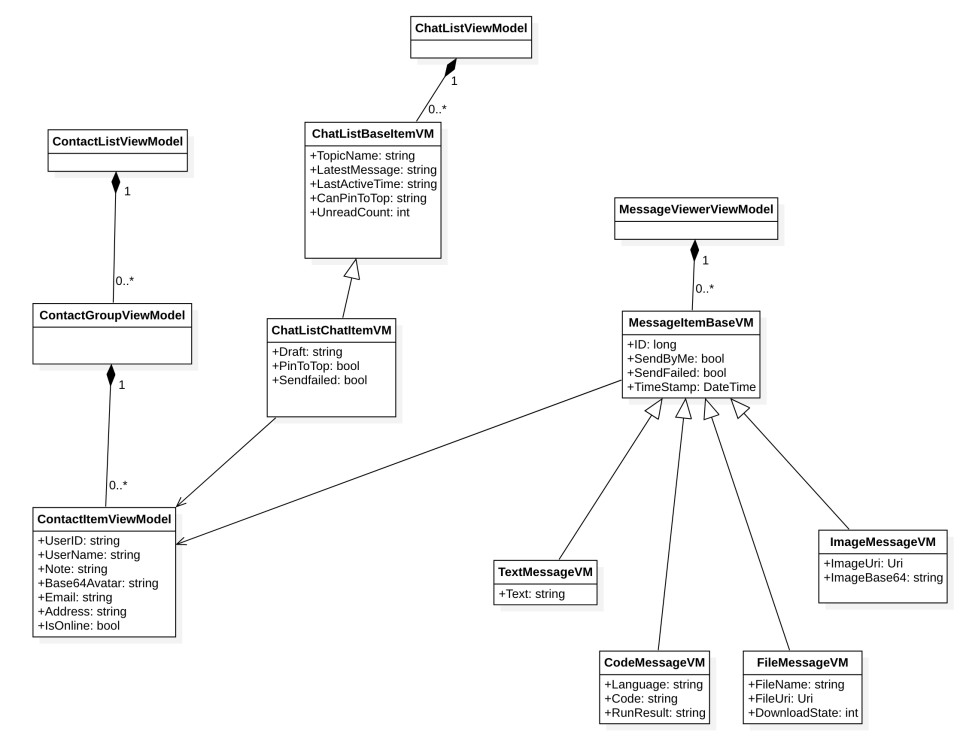
图**4.3.1** 界面逻辑层次结构

图4.3.1显示了界面主要组成部分及其层次结构。注意此图展示的仅是逻辑上的界面层次结构，实际代码中对应的类不一定有图中显示的父子关系。

**4.3.3关键的ViewModel**

MVVM模式中使用数据驱动界面，这里列出关键的ViewModel类来说明客户端界面主要展现的数据。不重要的属性已省略。

图**4.3.2**  关键的ViewModel

其中，ContactListViewModel为好友列表提供数据；ChatListViewModel为聊天话题列表提供数据；MessageViewerViewModel为消息浏览器（查看聊天消息的组件）提供数据。

**4.3.4关键功能实现**

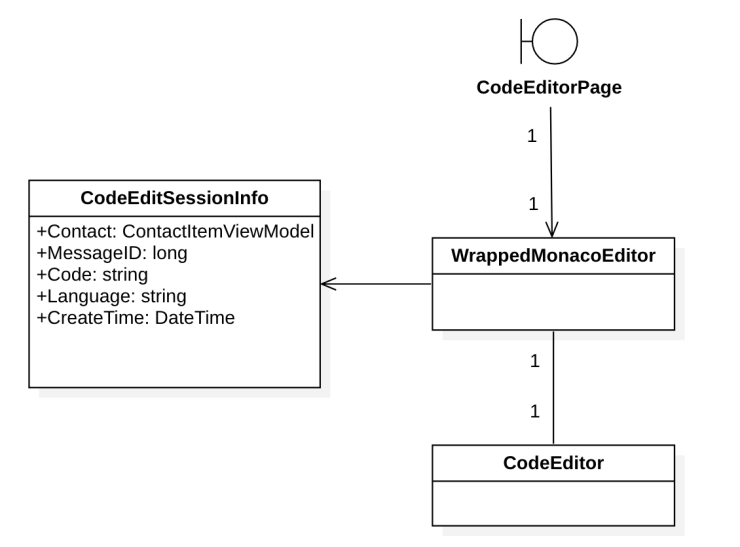
**4.3.4.1消息浏览器**

这里的“消息浏览器”查看聊天对话内容的界面组件。现在的即时通讯软件一般都采用一个个“气泡”的形式来显示消息，本软件亦是如此。这样的界面所用的技术在各个平台上都是相似的，就是使用listView并自定义其每一个项。但这类软件发展到今日，这样一个界面也并不简单：消息数量可能很庞大，消息数据到来的顺序有可能是混乱的；消息气泡的种类多种多样，比如QQ中有红包气泡、分享气泡等等；气泡可能会变化，甚至可能消失（撤回消息）；气泡间可能存在关联，比如“引用”这样的功能。

经过考虑，我们有选择性地实现了上述功能中的部分。每种类型的消息对应一种类型的“气泡”控件，共四种“气泡”控件：文本消息气泡、代码消息气泡、图片消息气泡、文件消息气泡。再使用平台框架提供的ItemTemplateSelector类，即可根据消息类型选择对应的消息气泡展现在界面上。这样做方便了后续的拓展，也使得消息气泡本身就像其它自定义控件一样具有提供各种交互操作的能力。

**4.3.4.2代码编辑器**

为了达到预期效果，我们对代码编辑器进行了多层封装。如下图所示。

图**4.3.3**  代码编辑器层级结构

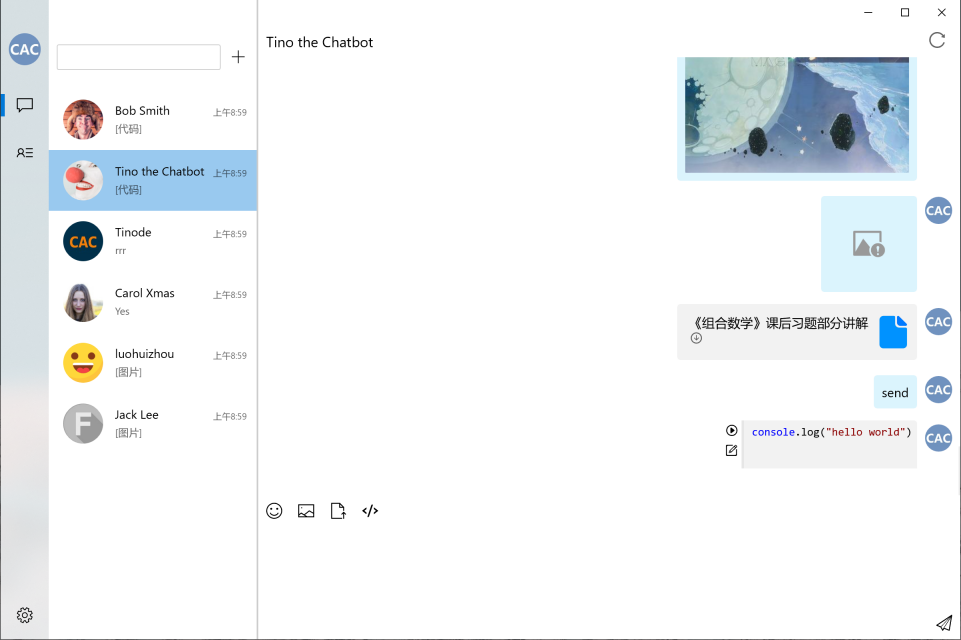
CodeEditor使用WebView对Monaco Editor进行了封装，实现了C#代码和js代码间的通信。WrappedMonacoEditor对CodeEditor进一步封装，提供了一些本软件需要的功能，如语言切换，运行按钮，运行结果显示框等。CodeEditorPage使用WrappedMonacoEditor，并增加了多标签页切换的功能，方便用户同时编辑多份代码。我们将一个标签页称为一个“Session”，并使用“CodeEditSessionInfo”类记录每个Session的情况，包括这个Session来自于哪条消息、创建时间等。

实现多份代码（Session）的切换，需要实现Monaco Editor的model（存储了代码、编程语言标记等的结构）切换。这需要在html文件中编写js代码，并通过webView的InvokeScriptAsync方法起调html文件中的js代码。

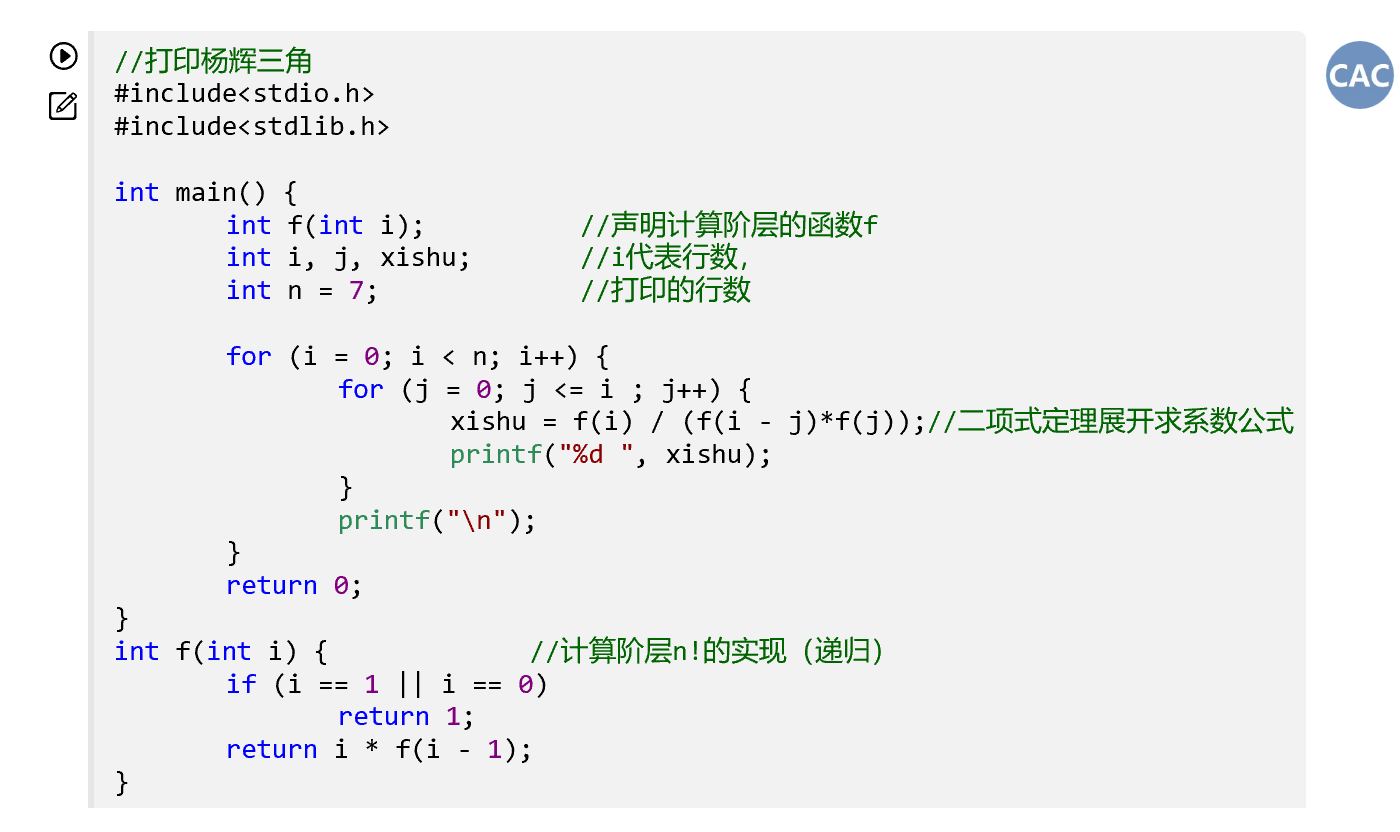
4.4 UI设计

本软件采用切合UWP平台风格的扁平化设计，设计时希望界面简洁、美观、易用。软件界面总体采用Master-Detail布局结构。左侧为功能导航及聊天列表（或联系人列表），右侧为聊天消息区域和消息发送区域（或联系人详情区域）。打开代码编辑器时，界面宽度将拓展，同时代码代码编辑器出现在最右侧。构建UI的过程中使用了一些UWP平台的特性，如资源字典文件和字体形式的矢量图标。这些特性本身也为构建良好的UI界面提供帮助。

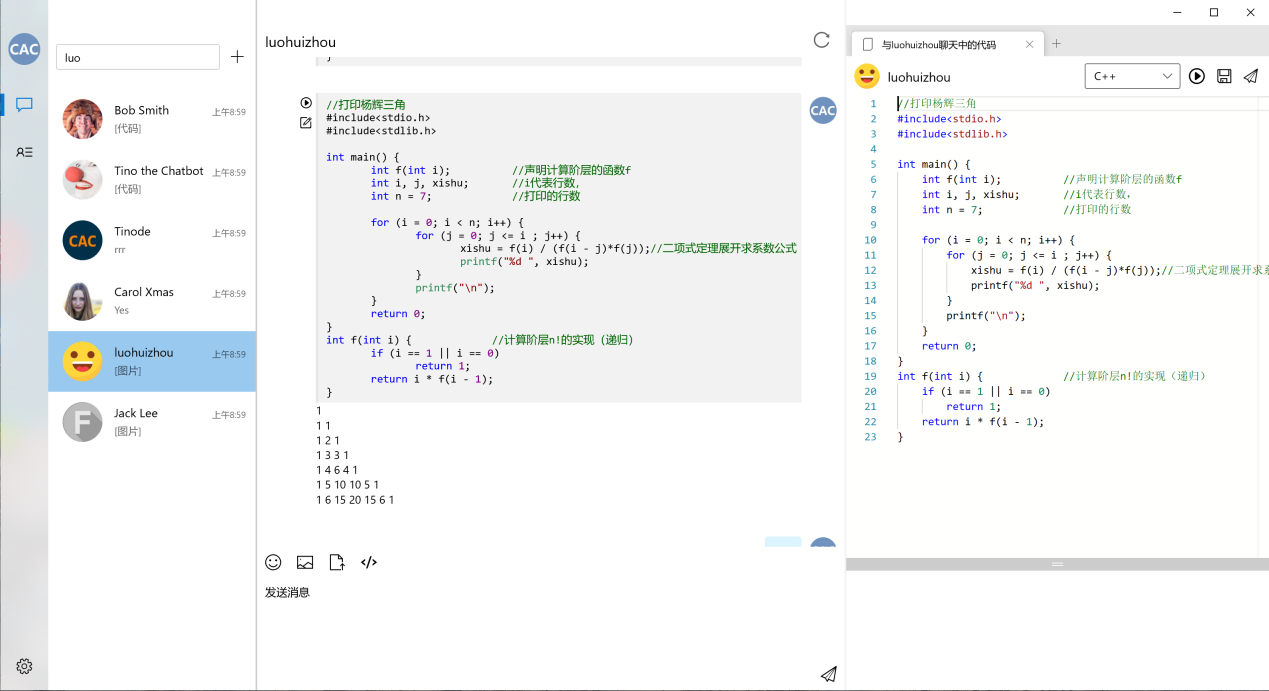
下图为典型的聊天界面。

图**4.4.1** 典型的聊天界面

针对代码的发送和接收，本软件提供了优化的显示方式。若一个聊天会话中有代码，它将显示在一种特别的代码高亮气泡中，如图4.4.2所示。

图**4.4.2** 代码气泡

编辑代码时，界面布局如图4.4.3所示。当用户屏幕宽度较窄时，代码编辑器将在新窗口中打开。

图**4.4.3** 编辑代码时的界面布局