### 项目简介

本项目是主要实现一个聊天软件，一般聊天软件提供的业务内容大概是文字聊天、语音/视频聊天、文件传输、多开同时会话、可发送表情及动画、保存聊天记录、创建群组等七大功能。而本次我想实现的是其中业务上一小部分的内容，将重心放置到UI结构设计、软件架构、传输交换协议处理、以及用户交互上来。

该项目只提供了登录与账号注册功能、发送文字与发送表情包功能、传输文件功能、添加好友功能、保存聊天记录功能，并搭建可在互联网通信的服务器。

#### 协议简介：

既然要实现网络聊天功能，网络间的通信协议便是不可不谈的问题，在TCP/IP四层概念模型中TCP/IP协议族的UDP和TCP是传输层协议不可或缺的重量级选手。其中UDP提供了无连接、尽最大努力交付、面向报文、是快速且不安全功能，其特点就是快速，且是无连接无反馈的快速发送，但缺点也是无连接造成的保报文丢失等不可靠因素，需要应用层实现可靠通信。在网络环境良好的区域上，建立在可靠的UDP协议基础之上的应用协议将得到非常优秀交换体验，而当网络不稳定时，应用层实现的可靠和重发功能往往会将UDP的快速优势清扫的一干二净，速度比TCP还低数倍。当然在实时内容上，UDP一般会采用不完全可靠的实现方式，及加快来传输速度，也在大量报文丢失时跳过重传来保证视频的实时性，如直播视频等大流量实时画面。

UDP固然在速度上有很多优势，但现在的网络确实大量基于TCP实现，其原因就是在复杂的互联网环境中，报文十分容易丢失而很多内容都是不容忍丢失的，如文件传输等。TCP提供的面向连接、提供可靠交付、面向字节流、稳定且安全可靠协议解决了应用层的零丢失问题，所以TCP协议才会被大量使用。尽管TCP协议需要三次握手建立连接，四次挥手的操作，而且是面向字节流，在应用层开发上会存在粘包以及压栈等问题，不过凭借着在传输层的滑动窗口的拥塞控制、全双工的通信连接减少了应用层对于网络流量的管理，而且使用连续ARQ的重传机制也不是慢到那里去，由此看来TCP协议被广泛采用不无道理。

回到聊天软件的实现，TCP协议固然有很多好处，但是这个协议确实针对传输层开发，对应用层使用的是流传输，存在大量数据时的粘包问题，如果只是单点通信那这个问题也不会是问题，而用于服务器上则可能会存在堵塞线程的情况，给应用层的处理添加了不少麻烦。所以本次将使用WebSockect作为通信的网络连接协议。

#### WebSockect协议：

WebSockect协议是建立在TCP基础之上的全双工长连接协议，主要用于浏览器与服务器的通信。在介绍WebSockect协议时，应该先介绍一下HTTP协议。HTTP协议最开始是出现在浏览器上的一种基于TCP的应用层协议，该协议的主要任务就是用浏览器向数据服务器发送请求报文，服务器根据报发内容反馈内容并马上断开连接。所以HTTP协议的特点就是简单快速、灵活且无连接、无状态的一种短时TCP连接。其中无连接无状态指的是HTTP建立了请求/响应模型来对服务器进行通信，一旦本次内容提交并由服务器反馈完成则不保留如何连接记录，浏览器想保存信息就必须重新发送，而这就是浏览器会使用Cookies记录并重发的原因，而Cookies尽管只是保证了客户端的信息记录，对于服务器来说依旧存在忙点。相比于HTTP的短连接、快释放的方式，WebSockect协议则采用了长连接有状态的通行方式，使得完成一次请求任务之后客户端与服务器之间依旧保持连接，WebSockect的出现将改变HTTP由服务器下发通知到客户端时只能采用轮训的方式，而是直接与已连接的客户端通信，极大地简化了服务器对浏览器的任务下发，同时减轻了服务器的资源开销。为什么该项目会采用WebSockect协议作为通信媒介呢，这是因为WebSockect与TCP在应用层的使用极为相似，且保留了将来在浏览器实现聊天软件的一种可能。

### 开发工具前置知识简介：

#### 客户端：

该项目使用C++的Qt库配合QtCreator工具开发客户端的UI界面，Qt库虽然是C++中的一部分，编程语法上却不是标准C++的规范，而是在C++的语法之上扩展出来信号槽的概念，在拥有C++的基础之后请务必对Qt的信号槽机制有所了解才行。Qt的各类事件消息依靠信号发送信息，槽接受消息，信号与槽之间没有关联，依靠QObject类提供的connect信号槽连接函数将信号与槽之间建立起连接，否则两者完全独立。信号槽的使用可以有效的降低模块编程的耦合性，但信号槽在用法上由三点限制。第一，声明一个信号的类必须继承QObject基类；第二，类内必须声明Q\_OBJECT宏才可以使用信号槽机制；第三，编写的Qt信号槽类必须由Qt提供的Meta-Object Compiler元模板编译器先编译出带moc\_前缀的C++文件之后才能进入到正常的编译流程。这三点限制算是Qt的一个特点，在Qt中所有的Qt界面控件都继承自QObject类，由QObject类提供connect函数实现观察者模式，将子类的信号与槽进行连接，当信号发送时通知槽函数。Qt中大部分事件信息都是用类信号发出，接收时只需定义槽函数与控件类信号连接并将功能在槽函数中实现即可。Qt的所有界面类都是继承于QObject，而这些类的事件函数绝大多数都是虚函数，这使得由子类重写事件函数并重新实现功能成为了可能。这是Qt带来的高度可定制化的一套模式，不管是如何一个界面类控件都可以重新继承并重写，从而实现定制化的UI展示效果。

#### 服务器：

本项目的服务器主要使用WebSockect协议通信，并部署在云服务器Linux系统的发行版ubuntu系统服务器版本上，与互联网连接并开放端口提供客户端连接。服务器所有用的网络编程技术是Linux操作系统提供IO复用技术epoll。

epoll是Linux内核为处理大批量文件描述符而作为select和poll的改进版本，它能显著提高程序在大量并发连接中只有少量活跃连接时的网络性能，主要用于解决网络IO编程中频繁操作系统提供的select/poll函数受文件描述符限制而无法满足高性能的情况。因为epoll中得到系统的事件支持，当连接无事件被记录在连接池中的sock并不会被加入到轮询容器中，只有触发了事件的连接才会进入轮询，避免了select/poll中无事件sock依旧进入轮询导致连接数增多会严重增加IO轮询时间，最后服务器因为轮询时间过长而导致新连接丢失或者任务超时的问题。

其中epoll接口提出的主要特点是，在一个进程中可以监听大量的文件描述符，而不会出现调用select或poll的最大描述符值的限制，一般select或poll的最大限制最佳数是1024个sock，而且epoll也支持高效地进行添加，修改和删除文件描述符的操作。

epoll的内部维护一个epoll的实例的文件描述符的变更信息并触发事件, epoll提供了水平触发模式和边缘触发模式，当数据量巨大时采用边缘触发提高IO性能，从而提供了更好的控制。当检测到文件描述符有数据变更时，epoll会以事件的形式通知应用程序。

IO复用在网络编程中非常重要，依靠单线程连接处理以及少量线程池就可以处理大量的高并发连接与数据交换，减少系统的线程开销就是变相提高系统资源的可用性。

服务器上除了不可忽视的IO复用技术之外，数据库的使用也尤为重要。服务器上的数据库用于存放用户信息，而用户信息量会随着时间的推移而累加，虽然该项目的数据量相对较少，但如果采用文件的方式存放并频繁的对数据进行增删查改的话，结果在文件操作上必然是服务器性能的瓶颈，而数据库对数据得操作速度则避免了这个问题。该项目采用了SQList3的轻量级数据库，数据库拥有轻量级的，无服务器连接的轻量级可携带等优点。SQList3支持标准的SQL语言，采用无数据类型，所以可以保存任何类型的数据，支持标准数结构类型，而且是无配置的，只生成一个.db的数据库文件，将该文件带到哪里，哪里就是数据库。其便携性可以在多地、不同设备开发而不用考虑数据库数据迁移问题，是轻量级项目的最优选择。

#### 网络库:

建立通信连接时固然可以自己直接使用C库的sockect()函数API，不过在实际开发中都会采用网络库，或者重新编写一个网络库，绝不会单纯的使用sockect套接字，因为该函数是提供了最基础的数据交发送功能。其中网络编程中存在的几大问题，TCP粘包问题，非常常见，因为是TCP是面向流字节传输的；UDP的报文丢失问题，使用UDP是必须在应用层实现可靠的重发策略；TCP的长连接问题，虽然TCP底层已经存在保活设置，不过要是由于硬件设备的突然中断所导致的连接断开则不反馈，所以应有层需要实现自己的心跳来处理意外断开；流量控制问题，虽然TCP提供来滑动窗口来做拥塞控制，但是确是基于路由器交换的，如果如果是上层应用的不当操作所造成的TCP数据压栈会影响其他进程的无法发送数据，所以要对数据发送的速度做限制，以避免应用层的数据大量发送，达到流量控制的目的；断线重连问题,TCP是一个长连接的协议，但是网络却不是永远都这么好，当网络发生错误导致TCP套接字断开时应用层应该考虑TCP的重新连接以达到用户友好的目的；通信加密问题，TCP采用的是明文传输，在数据安全性上未留下余地，导致重要信息被轻易获取，所以在互联网中传输数据时安全性不得不好好考虑一番；大小端问题，由于硬件的不统一，在内存的硬件存放上存在大端、小端两种不同的内存数据正反拜访顺序，它们的数据在内存中按字节端相反，如果没有对大小端进行处理则会读取到与预期不符的数据；多线程顺序问题，在客户端中发送消息通常只有单线程发送与接收，但在服务器上由于到同时处理多个sock连接的数据，所有都会提前生成线程池等待数据处理，而多线程在发送或者接收时可能会因为时间并发，如果读取的返回时间不一致会导致先读的后返回，从而导致获取的内容顺序错乱。串包问题，这个问题是由原本发送请求的连接因为网络原因断开之后其原本的sock被新连接的数据占用，而服务器接收到请求并进行反馈时却将反馈内容返送到了新连接的sock上，导致断线重连的sock未能接收到反馈包，而服务器也错误的将包发送到其他sock的错误。

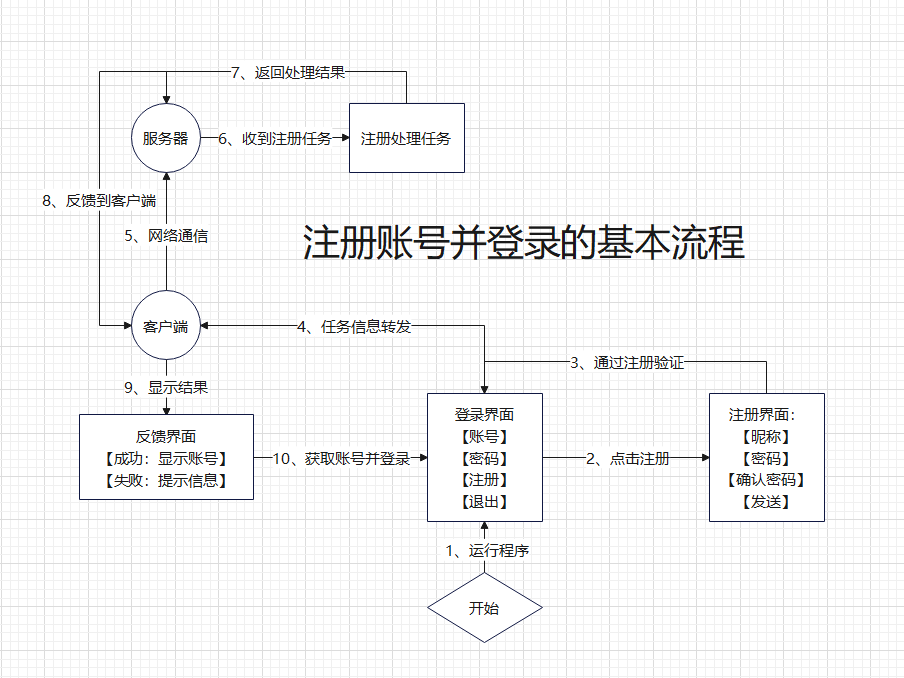
认识到上述网络编程可能存在的问题之后，选择一个合适网络库将可以避免错误发生的同时减少开发时间，减轻了网络编程的可靠负担。在几大热门libevent、libev、libuv、libhv网络库中，选择了libhv网络库作为本次项目的网络通信开发。libhv库是近几年才兴起的网络库，因为其简单易用，且提供C++接口；提供SSL/TLS的加密通信增加安全性；提供心跳、重连、线程安全等编程接口；提供装包、拆包模式；提供HTTP和WebSocket协议支持；提供Linux、Widows、MacOS、Android、iOS等常见操作系统的跨平台支持；提供IO复用事件循环等众多优点，使得在Github开源社区上拥有越来越多的使用者和维护者。

## 基本流程

经过前面的前置基础知识介绍之后，接下来要将重心转移到到整个软件的设计架构、UI衔接以及任务处理的流程上来了。

### 登录流程：

#### 注册流程图介绍：

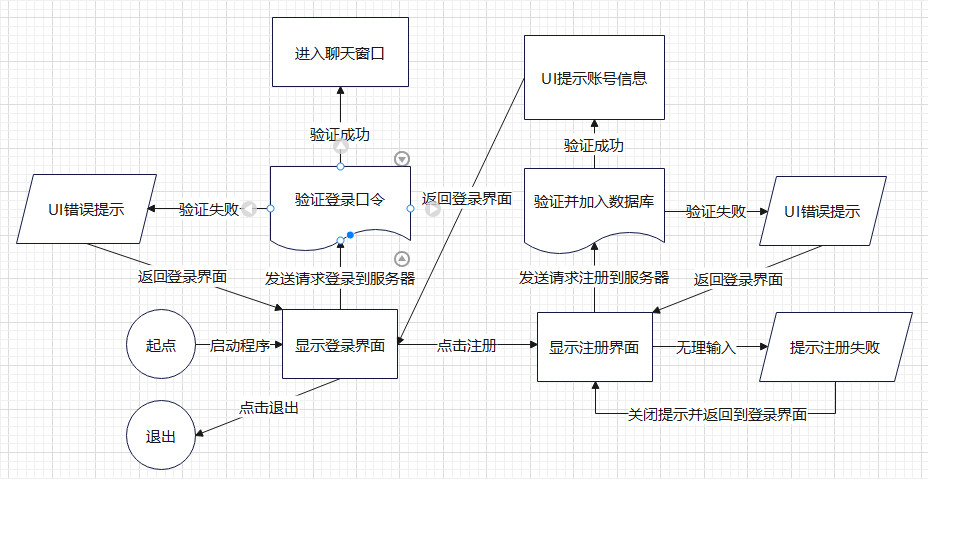


图表 注册流程图

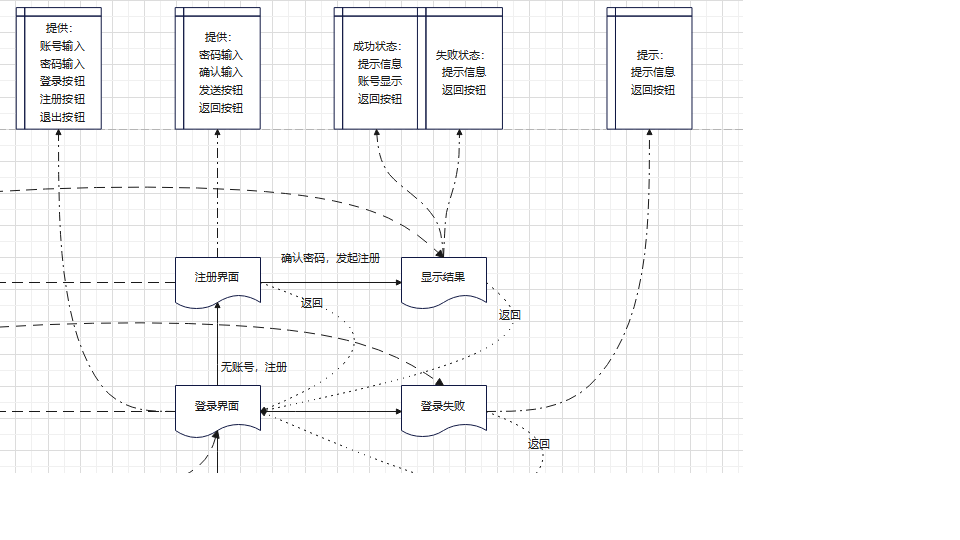
中流程图中看出从无账号状态，在软件中注册账号并成功登录到程序界面。可以看出注册界面与登录界面之间的关系，注册界面为登录界面的子类，注册界面的所有操作都会反馈到登录界面，再有登录界面反馈到管理类。管理类在初始化中已经建立与服务器了WebSockect的网络通信，最后注册界面的信息会由管理类发送到注册界面。

服务器在网络连接中接收到了客户端的信息并判断协议类型进行处理，处理结束之后反馈到客户端，此时再有客户端接收服务器的反馈信息，根据反馈的成功与失败在界面上做出相应的提示。在客户端的响应中，不管是成功或者失败最终都会重新回到登录界面上给用户。

#### 登录状态图与控件图介绍：



图表 UI切换状态状态图

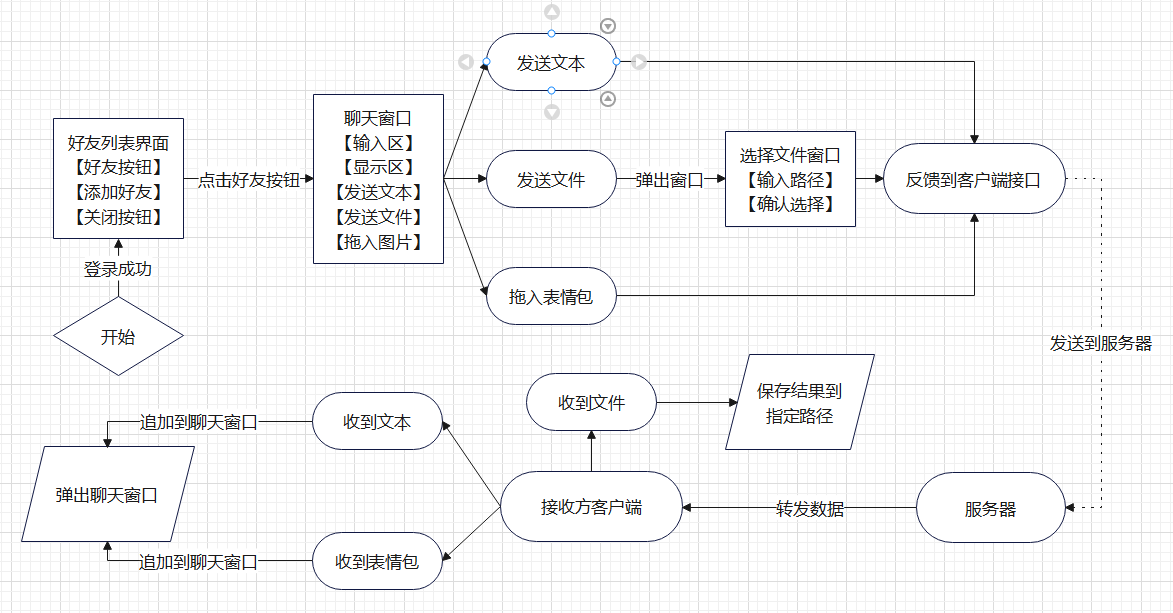


图表 登录界面UI控件图

从状态图可以看到各种任务的成功失败与UI窗口的反馈关系，状态图简化了网络之间的通信连接，可以更加直观的看到不同操作下成功或者失败之后操作与对UI的反馈与切换，其状态图的反馈到成功登录并显示好友聊天窗口为止。关于在状态图中的各种UI显示细节在UI控件图中都有显示，其中列出了各个UI窗口输入与显示所使用的控件细节。

### 交换流程：

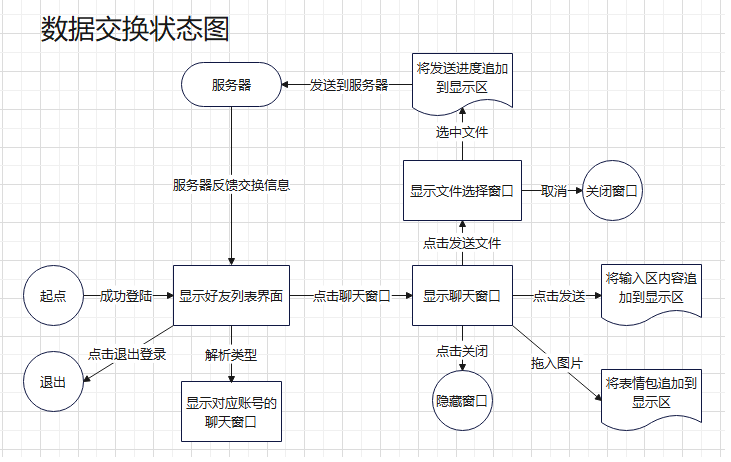
#### 交换流程图介绍：



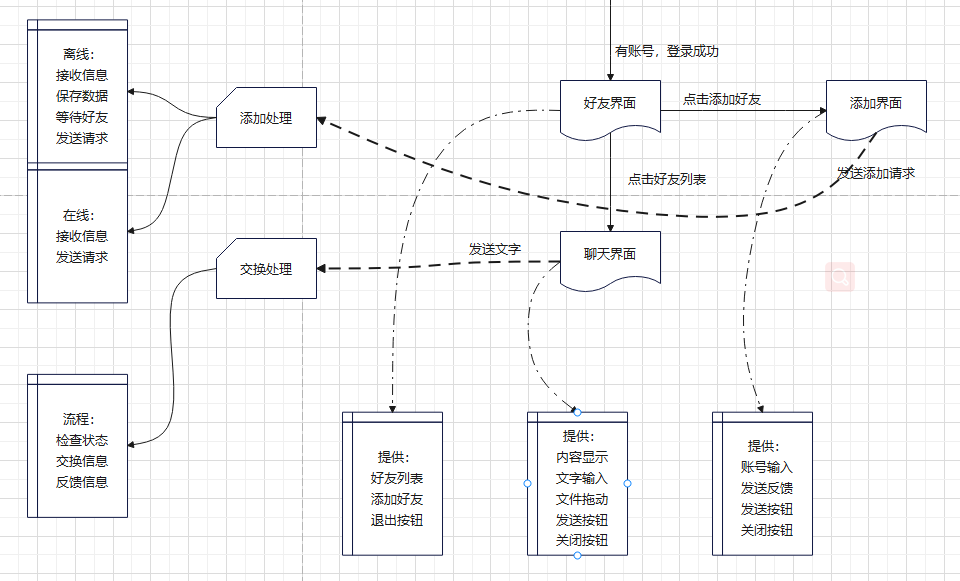
图表 交换流程图

在完成注册和登录流程之后，接下来就进入来正式开启聊天的过程。聊天的第一步就是要将自己的的信息成功的发送到另一个程序上来，而交换流程图就是试图去将这个过程以图形的方式呈现出来。从交换流程图中可以看出，登录成功之后的好友界面提供了好友按钮列表，该列表上拥有已经成功添加的好友，点击好友按钮就会弹出对应的聊天窗口。聊天窗口分别提供来三种输入方式，发送文本、拖入表情包和发送文件。其中拖入表情包是无需确认的，拖入即发送，发送文本是等待确认键按下后发送出已经输出在输入窗口的文字，而发送文件则会提供一个窗口让用户选择想要发送的文件，如确认要发送的文件之后，在显示区就可以看到发送文件的进度条信息。其中，不管是发送了哪一种类型的信息，其信息都会立刻追加到显示窗口，而接收方会在收到服务器的转发显示时解析出对应的信息类型并显示在他的聊天窗口上。如果没有任何动作的情况下发送聊天窗口突然弹出来了，不要惊讶，一定是有人给你发消息了。

#### 交换状态图与处理图介绍：



图表 交换状态图

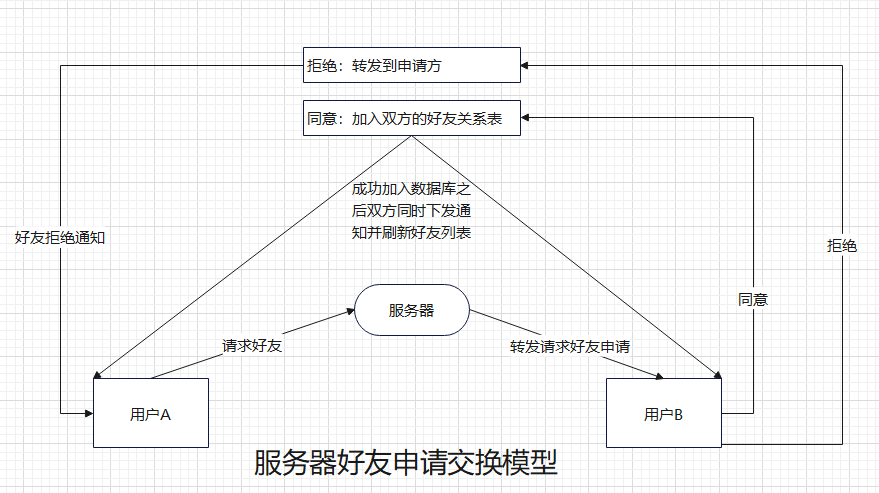


图表 交换处理图

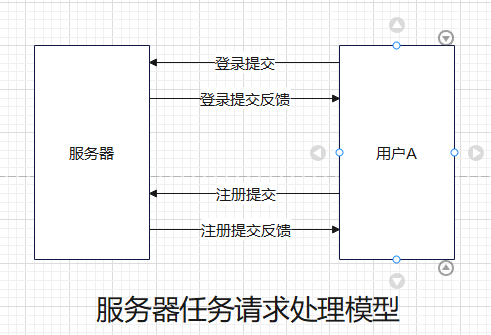
从状态图中可以清晰的看出交换数据时的各个UI控件的转换，在点击好友按键之后，会弹出对应好友的聊天窗口，而接收到服务器的交换信息之后也会弹出对应的好友聊天窗口。在接收方的交换信息的显示中，有一点需要注意，发送文本与发送表情包都是在全部数据接收完成之后一次性完成显示的，而发送文件则会将信息以进度条的方式持续显示，这是因为文件一般都比较大，短时间内无法完成交换，在本项目中并没有限制发送文件的大小。

在服务器的交换处理图中可以看出，因为不确定在发送信息时对方是否可以马上接受，所有设置了发送缓冲区确保了当好友不在线时依旧可以发送信息，而好友上线时去检索离线缓冲区内是否有发给他的信息，如果有则可以在上线时接收到信息。

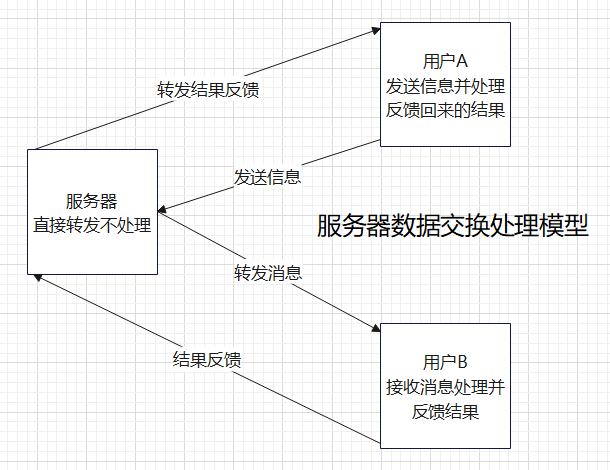
#### 添加好友流程：



图表 好友添加处理模型



图表 任务请求响应模型

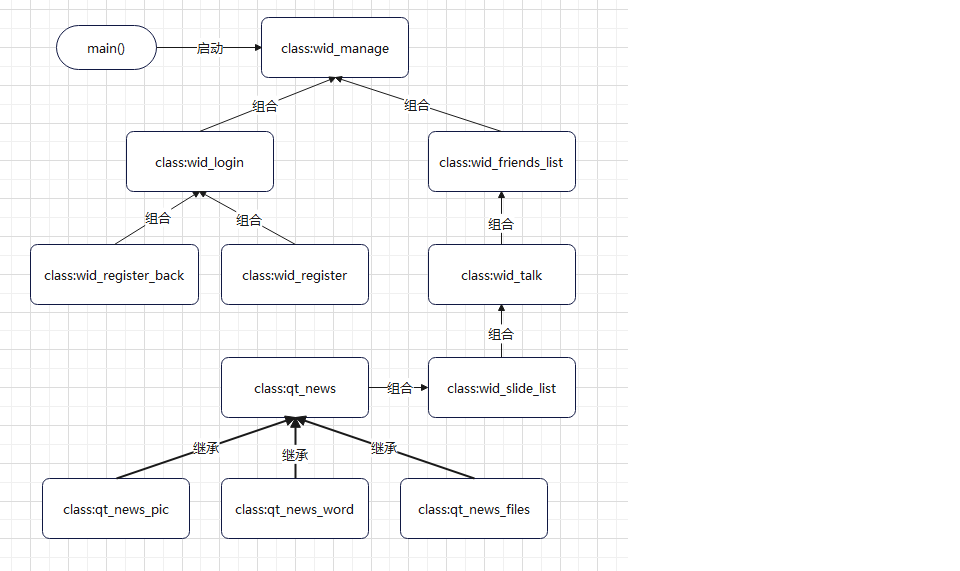


图表 数据交换模型

好友模型在数据转发到另一个客户端时对数据进行了处理，既设计到对数据的处理也涉及到数据的转发，所以从服务器的三个交换模型中可以看出，好友添加的处理模式是较为复杂的。因为好友添加时涉及到来对另一个用户的请求答复，而答复的结果又会影响到服务器的行为，所以服务器在转发时做了更多处理。与数据交换模型相比，好友添加模型多了服务器对数据的处理，因为数据交换在服务器不需要处理任何数据，只需要直接查询转发即可，故而简单；与请求响应模型相比，请求响应模型只需要将接收到的数据进行处理并原路返回结果即可，不涉及其他用户客户端，故而简单。好友添加处理模型的设计其实是为了优化用户体验而单独设计的，因为在成功添加好友之后服务器需要同时向双方下发反馈结构，提示客户端重新拉去好友列表的信息，将新好友及时更新到好友列表中。添加好友处理模型并不是必须的，因为将数据交换模型与请求响应模型结合在一起使用也可以达到一样的目的，只不过需要在客户端做更多的处理，也因为多次交换而浪费更多网络资源，故而采用新的处理模型优化处理流程。

### 界面篇：

#### 类关系简介：



图表 UI类关系图

从关系图中可以看出，wid\_manage类组合两个子类wid\_login类与wid\_friends\_list类，wid\_login类负责登录之前的注册操作，而wid\_friends\_list类，负责登录之后创建聊天窗口。wid\_friends\_list类下的成员wid\_talk类负责将内容展示到界面，展示的内容又交给wid\_slide\_list类管理。其中wid\_slide\_list类是一个可滑动的列表窗口，将QWidget的子类加入列表并提供滚动的展示，其中类内存在一个容器记录来qt\_news类的指针，而且qt\_news类是虚类，不提供实例，wid\_slide\_list类管理的所有的内容都由继承自qt\_news的子类提供。

#### 管理类介绍：

main函数执行的第一个流程是wid\_manage类，wid\_manage类并不是一个图形界面类，自身并不会显示图形界面，但是却控制着整个客户端的运行流程。wid\_manage类作为全局管理类控制整个程序的登录、注册以及登录成功之后的界面显示。wid\_manage类控制登录界面和登录之后的聊天界面之间的切换与隐藏，wid\_manage还是控制客户端与服务器连接和通信的唯一桥梁。登录之后的操作依旧由wid\_manage类控制，网络信息全部由该类统一分发与接收，这样做的好处是统一管理了所有信息的处理流程，中心化操作与任务分发，容易编写出模块化的组件配合工作，且不必关系其他类的实现，统一由管理类进行调度，实现来模块化之间的解耦操作。不过中心化管理的缺点是信息处理传输不够灵活，而且多层次的信息传输需要转发，造成信息转发工作量增加。

#### 登录类介绍：

从图中得知，wid\_manage类的成员wid\_login类是一个图形界面类，提供客户端的登录界面。登录界面中会提供账号输入框、密码输入框、确认登录按钮、账号注册界面、退出登录按钮。



图表wid\_login类

在无账号下wid\_login类提供账号注册wid\_register类与wid\_register\_back类，wid\_register类需要申请者提供密码，如果注册成功则由服务器返回一个一组被登记的账号信息，该信息由wid\_register\_back类展示给用于。

#### 注册类介绍：



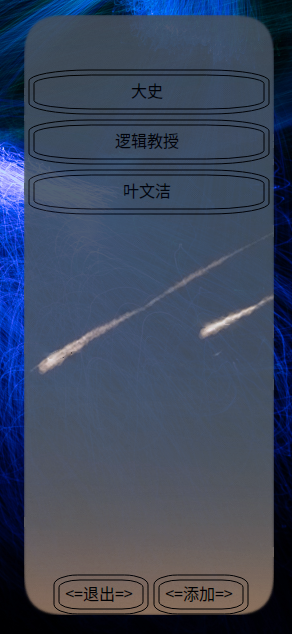
图表 wid\_register类



图表 wid\_register\_back类

注册类为登录类的成员，在点击注册之后wid\_register类将发送一个附带着昵称和密码的信号给wid\_login类，接收到信号之后再将该内容转发给服务器，服务器处理结束之后反馈信息到wid\_login类，在将反馈信息交个wid\_register\_back类完成整个注册流程。

#### 好友类介绍：



图表 wid\_friends\_list类

登录成功之后就进入wid\_friends\_list类的界面，该类记录本账户的所有好友，并提供一个好用列表按钮组。wid\_friends\_list类的按钮容器记录所有按钮以及对应的好友账号，点击按钮是会触发对应账号的wid\_talk类聊天窗口，同时在窗口下方放置了添加好友和退出登录的两个按钮。

#### 聊天窗口类介绍：



图表 wid\_talk类

wid\_talk类由两大部分组成，分别是输入区qt\_edit\_text类和显示区wid\_slide\_list类。qt\_edit\_text类由Qt控件QTextEdit接收键盘内容到文字缓冲区，QPushButton控件确定发送文字缓冲区到显示区。wid\_slide\_list类是一个长度可变但是只能通过固定大小窗口预览的内容显示窗口，该类由QScrollArea和QWidget两个控件组成，QScrollArea将QWidget设置成一个可滑动区域，当有内容不断写入显示区，QWidget控件长度被拉长时，即可达到窗口滑动预览的效果。

wid\_slide\_list类在内部有一个vector的容器成员，记录着所有加载到显示区的内容。当文字被加载到显示区时，被加载的内容是一个QWidget界面而非单纯的文字，所有被加载的控件都必须从qt\_news 类继承，并重写to\_string\_info纯虚函数以获取历史记录，显示文字的继承类为qt\_news\_word类。

显示区继承qt\_news 类的设计理念是因为显示区需要回收记录记录，而发送到显示区的不仅是文字还有图片、视频、图片、表情包、动态包、特效等多种多样的展示，不过目前仅提供来三种展示效果，分别是:qt\_news\_pic类显示表情包、qt\_news\_word类显示图片、:qt\_news\_files类显示文件传输状态。如果局限于文字将无法获取良好的展示效果，而区展示的内容是QWidget类则完全不同，QWidget控件内可以随意填充各种想要显示的画面将带来非常大的灵活性，同时强制重写to\_string\_info转字符内容的纯虚函数为各类型的历史记录保存带来了统一的接口，在显示和聊天内容回放上都十分灵活。

## 通信篇：

在通信篇中会介绍整个程序是如何将信息层层传递出去的，因为在实际的开发中，将一条信息传递出去并不会和想象中的简单，因为要考虑到整个系统的模块化。模块化系统可以降低整个系统的复杂度，可以解耦不同模块之间的关闭，使得其他模块无需考虑别人在做什么，只需要做好自己的事情便可以由全局控制的管理类掌控整个程序的流程。在该项目上也会尽量的使用这种编程方式，不过因为整个项目的体量不够大，有时模块化编程会给人造成一种多余的感觉，但这些多余大体量的项目和多人联合的实际开发来说是非常有用的，该项目给出了一种解决方案。在这一篇开始请先具备基础的编程知识，因为从这一篇开始会以程序员的视角出发，进入到实际的编程环境当中。

#### 程序入口：