

南京邮电大学通达学院

毕业设计（论文）

题 目 基于 Android 的聊天系统的设计与实现

专 业 软件工程(嵌入式培养)

学生姓名 项伟伟

班级、学号 18240125

指导教师 王俊；余西亚

指导单位 中兴软件技术（济南）有限公司；

南京邮电大学通达学院

日期：2022 年 3 月 7 日至 2022 年 6 月 10 日

摘 要

如今互联网技术的发展，各个平台的数据量正在不断膨胀。完全本地化编程已经越来越少，适应网络编程已经成为程序员的必修课程。本文通过设计一个基础的即时通讯软件浅谈 TCP、Socket 网络编程和软件体系结构设计，分析实际运用中的解决方案。

本设计采用 C/S 架构，使用 Kotlin 编写基于 AndroidQ 的客户端。使用 Python 构建跨平台的服务器。服务器主要承担用户基础数据和消息暂存的功能，数据最终流向为客户端到客户端。软件整体设计基于 TCP 协议，使用 Socket 进行编程，主要涉及自定义协议和应用层架构设计。数据库采用 MySQL 和 SQLite，前者用于服务器，后者主要用于客户端数据存储。整体采用面向对象和面向服务编程，本系统具有较高的可维护性和可拓展性。

关键词：Android；Kotlin；Python；TCP；Socket；软件工程；

ABSTRACT

Nowadays, with the development of Internet technology, the amount of data on various platforms is expanding. Completely localized programming has become less and less, and adapting to network programming has become a compulsory course for programmers. This article discusses TCP, Socket network programming and software architecture design by designing a basic instant messaging software, and analyzes the solution in practical application.

This design adopts C/S architecture and uses Kotlin to write Android Q-based client. Build cross-platform servers with Python. The server is mainly responsible for the user's basic data and message staging functions, and the data ultimately flows from client to client. The overall design of the software is based on the TCP protocol, using Socket for programming, mainly involving custom protocols and application layer architecture design. The database uses MySQL and SQLite, the former is used for the server, and the latter is mainly used for client data storage. The whole system uses object-oriented and service-oriented programming, the system has high maintainability and scalability.

Keywords: Android; Kotlin; Python; TCP; Socket; Software Engineering;

目 录

| | |
|---------------------|----|
| 第一章 整体架构设计 | 1 |
| 1.1 简述 | 1 |
| 1.2 需求分析 | 1 |
| 1.3 服务器需求分析 | 2 |
| 1.4 环境依赖 | 3 |
| 1.5 概要设计 | 4 |
| 第二章 数据结构设计 | 6 |
| 2.1 用户消息盒子 | 6 |
| 2.2 自定义网络协议 | 9 |
| 2.3 客户端其他数据结构 | 10 |
| 第三章 客户端设计 | 12 |
| 3.1 UI 设计 | 12 |
| 3.2 通讯设计 | 14 |
| 3.3 资源管理系统 | 16 |
| 3.4 业务逻辑设计 | 17 |
| 3.5 锁机制 | 18 |
| 3.6 消息的传递 | 19 |
| 第四章 服务端设计 | 20 |
| 4.1 通讯设计 | 20 |
| 4.2 数据库设计 | 21 |
| 4.3 调度器设计 | 21 |
| 4.4 服务器交互逻辑 | 21 |
| 第五章 设计评估 | 23 |
| 5.1 功能评估 | 23 |
| 结 束 语 | 25 |
| 致 谢 | 26 |
| 参考文献 | 27 |

第一章 整体架构设计

1.1 简述

自即时通讯(IM)软件诞生以来,其便利性受到社会各界的青睐。典型的代表为微信、QQ等。即时通讯比传统电子邮件所需时间更短,且较之与电话更为方便。其主要特点为:多任务作业、异步、长短沟通、媒介转换迅速、高交互性、不受时空限制。

早期的即时通讯软件只能进行文本、预设的图片、文件的交流,依靠服务器进行缓存。如QQ等,其早期的会员功能可以让服务器长期缓存聊天数据,而后诞生的微信,则只采用服务器短时缓存的方式,非持久化保存聊天数据。伴随移动互联网的发展和Cov19的时代背景,即时通讯服务开始提供会议、VoIP。各种媒介的边界因为即时通讯而变得模糊。

另一方面,由于当今各大互联网企业的相关业务的发展,即时通讯软件已成为集生活服务、社交、娱乐等于一身的功能性软件,其冗余的功能备受争议,如QQ移动端集成了虚幻SDK(Unreal SDK)等。本设计将实现一个精简、小巧而纯粹的即时通讯软件,研究TCP协议和Socket编程。

1.2 需求分析

聊天App的定位是一个基于安卓平台的轻量化聊天软件。通过对当前存在的即时聊天如QQ、WeChat,和办公聊天钉钉、企业微信、网易POPO的研究和总结,设计并开发一款实现了用户基于账号登录注册,添加好友,发送文本,图片等基础功能的聊天软件。需要熟悉并掌握不同平台通信技术的底层原理及相关的代码实现及安卓平台、服务器端数据结构的运用以及文件存储和读取相关技术。项目整体基于C/S模式,客户端系统要求简洁且拥有优良的可视化图形界面。保证用户的使用流程尽可能简单、直接。其次,系统本身需要具有较高的可扩展性和可维护性。其次,由于安卓Q以上系统对用户隐私的保护,客户端应该逐级获取用户系统的访问权限,使用沙盒环境进行开发。对于服务器,服务器需要能够兼容各平台运行,并且要求其具有优良的扩充性和稳定性。

在广域网范围内,互联网通讯的主要瓶颈仍为带宽、线路。所以整体网络设计应该确保网络的可靠性。

1.2.1 登录注册

用户输入用户名和密码进行登录。需要对其进行检查和预处理,首先对用户密码进行哈希处理,构造传输协议,向服务器发送请求。服务器返回验证结果,当登录失败时需要反馈用户,当用户成功登录或注册,服务器需要传输基本的用

户信息。如 UID、昵称等。此后进入初始化阶段。服务器需要实现对数据库的访问、和信息登记。

1.2.2 应用初始化

应用初始化需要考虑从远端拉取联系人列表和服务器暂存的消息。在拉取的过程中，注意连接的有效性和用户网络状态可能的更改。

1.2.3 收发消息

通过点击联系人或已有消息进入聊天界面，双方可以在此进行图片、文本、定位信息的交流。其中消息界面应将最后接收的消息置顶，聊天界面需要将最后的消息置底。服务器不存储用户之间的聊天信息，仅作缓存处理。当某方发出消息后服务器应该提示另一方尽可能取走消息，避免占用服务器资源。

1.2.4 添加删除好友

用户可以通过输入对方用户名进行添加好友的，当用户 A 向用户 B 发起添加好友的请求时，检查是否已经添加，否则 B 会收到添加好友的消息。当 B 接收后双方即可在联系人查看到对方。

删除用户一经某方提出，数据库将直接删除信息。

1.2.5 日志

客户端使用安卓 Q 进行开发，而 Android Studio 提供了内置的日志工具类 `android.util.Log`。其提供了 `v`、`d`、`i`、`w`、`e` 五种级别的日志信息。其中：

`Log.v` 对应级别 `verbose`，用于打印繁琐的意义最小的信息，通常这些信息只是运行时的提示。

`Log.d` 对应级别为 `debug`，用于打印一些调试信息，这是非常常用的 `debug` 消息提示。

`Log.i` 对应级别为 `info`，用于打印一些比较重要的数据，这些数据往往是开发者想看到的，且能够帮助分析用户行为改进程序设计的信息。

`Log.w` 对应级别为 `warn`，打印警告信息。

`Log.e` 是最高级别 `error`，打印错误信息，异常信息、`catch` 内的错误信息都可以打印。

在项目开发过程中，往往较少使用 `System.out.println()` 或 `println()`，其原因是日志开关不可用、不能添加日志标签、日志没有等级区分等。其次，`Log` 拥有 `Logcat` 进行管理，可以方便的进行筛选、过滤。

1.3 服务器需求分析

1.3.1 数据库连接

当用户请求数据，如登录、拉取联系人等。服务器需要对数据库进行处理，需要设计用户信息表，用户关系表。

其次由于用户在登录、初始化界面需要等待，其性能会较大程度影响用户体验。

1.3.2 缓存机制

由于用户消息是较大数据量，故服务器不进行用户聊天数据的存储，需要构建一个缓存机制存放用户没来得及取走的消息。

客户端临时 IP 也是需要临时存储的数据之一，其原因是能较好的预知客户端位置，而非等到客户端主动联系服务器。

1.4 环境依赖

1.4.1 客户端依赖

客户端依赖于 Android Studio 集成开发工具。开发环境使用 Gradle7.3。

Gradle 使用了一种基于 Groovy 的领域特定语言来进行项目设置。Gradle 并不是专门为构建 Android 应用程序而开发的，各个项目仅需声明即可。

客户端代码使用 Kotlin 编写，Kotlin 是一种在 Jvm 上运行的静态类型编程语言。Kotlin 相比于 Java 最大的优势在于，其极大程度上避免了空指针的发生，且完全兼容 Java。其次 Kotlin 内置了非常多的面向对象编程实现。这是由于 Java 语言的特性，所有的 java 文件都将编译为.class 文件，而最终 Jvm 运行的是.class 文件，不在乎源码是什么语言，所以 Kotlin 能完美兼容 Java。

本次开发的目标平台是安卓 11，最低安卓 9。安卓系统作为用户量最大的，最成熟的移动操作系统，其在安卓 11 后不再允许应用在 sdcard 目录下建目录和文件，应用只能够往特定目录的私有文件夹写入数据。该版本加强了隐私保护、改进了通知系统，也包含了 AndroidStudio 4.x 的更新。新增的实时布局检视器 LayoutInspector 能够以 3D 形式展示界面所有元素与层级。极大地方便了开发者对 UI 的设计。其次数据库检视器，嵌入式模拟器，TF 模型导入，内存 Profiler 等突出了对系统跟踪器的界面改进。开发者可以通过 CPU Profiler，借助 vsync 查看每一个显示帧的时序来知晓卡顿的位置。

1.4.2 服务端依赖

服务端采用 Python 3.10 编写。Python 是一种动态解释型的编程语言，可以在任何安装 Python 实现的机器上使用，无论底层是由什么语言实现。当前网络编程的瓶颈仍在带宽，所以服务器性能可以稍微降低。虽然 Python 消除了很多面向对象的元素，但其仍具备较强的面向对象特性。其依赖社区非常活跃。至 2022 年 Python 已被逐渐广泛应用于系统管理任务的处理和 Web 编程。

服务器数据库使用 Mysql8.0，数据库连接池使用 PyMySQL。PyMySQL 是在 Python3 版本中用于连接 MySQL 服务器的一个库。

1.4.3 通讯协议依赖

协议采用 Google 的 Protobuf。Protobuf 是 Google 公司内部的混合语言数据标准。它可以用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。Protobuf 的是以二进制存储的协议，Protobuf 相比 xml、json、messagepack 在压缩比上有较大优势。在序列化和反序列化的速度上优于 thrift。但由于 Protobuf 需要进行预编译，在使用编译型语言时会造成较大麻烦，如 CMake 交叉编译。所幸本设计采用 Kotlin 和 Python 无需交叉编辑，非常适合使用 Protobuf。从接口定义的灵活性来说，messagepack 较 Protobuf 以及 thrift 较好，后两者都要预先定义 schema 并相对固定。

1.5 概要设计

1.5.1 架构设计

客户端主要分为三层，UI、本地、网络服务。UI 向用户展现各种信息，本地存储了图片等二进制信息，网络服务为整个 App 提供了信息服务。

服务器也分为三层，网络通讯层、逻辑处理层、本地资源层。网络通讯提供了监听端口、转发数据的功能，逻辑处理层为多样化的用户请求提供支持，本地资源层存储了二进制文件和数据库记录。按功能不同，进行层次划分，使各层功能相对独立。同时以调整调用第次，以达到层次之间的松散耦合。

在网络通讯的过程中，两者交互使用短连接，不记录状态。网络连接中，用户使用 UID(唯一识别编号)和临时 key 进行验证和通讯。客户端的表现层不应涉及到网络、存储等的访问，完全依赖于内存数据运行。在设计图片等二进制文件采用懒加载的方式。即开启一个线程让线程去本地取图片如果没有，线程自行访问服务器进行获取，如果服务器有，则进行下载缓存。无论结果如何，线程都将被销毁。其大致的拓扑图如下所示：

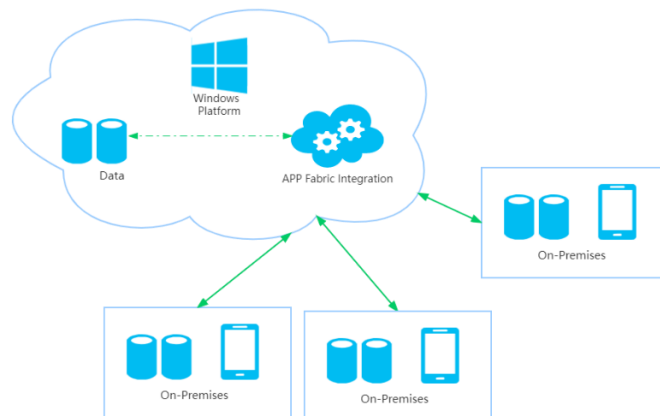


图 1-1 APP 网络拓扑图

在 APP 使用的过程中，用户只有登录以后才可以进行其他操作。

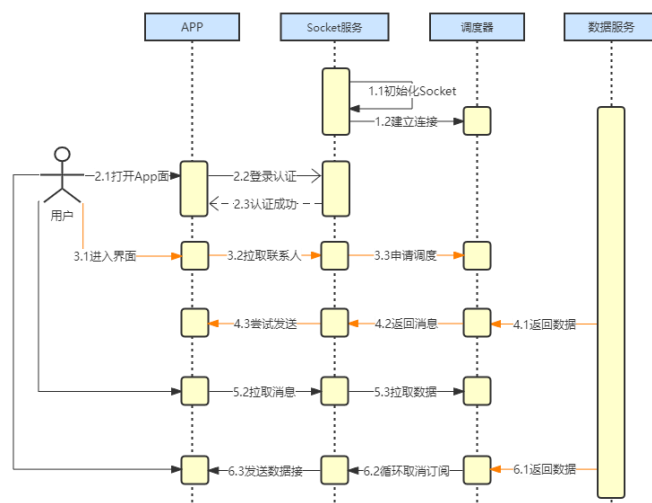


图 1-2 登录及初始化流程

如图 1-2 所示，用户登录成功后，需要拉取联系人列表，然后拉取消息列表。两者的时序存在相关性。主要的时序是：用户登录、服务器保存用户 IP 信息、返回用户 UID、用户拉取联系人列表、服务器检索数据库、服务器构造返回集、服务器尝试连接客户端、客户端接收联系人信息、客户端拉取消息信息、服务器检索消息池、服务端构造消息集返回、关闭连接。

第二章 数据结构设计

2.1 用户消息盒子

2.1.1 基本需求

常规即时聊天软件的消息界面并非采用传统数组或链表进行存储。其主要的需求是，当用户接收或发送消息后，该消息盒子需要置顶，以使用户查看或知晓。其次，安卓平台的 `RecyclerView` 组件在显示时需要根据元素的地址进行随机访问，所以数据结构的设计要表现出：能根据时间戳和数组下标进行快速访问、能够快速调整、且可迭代的特征。用户消息盒子即设计出来保存消息。

2.1.2 使用数组或链表实现

使用链表时，当用户接收到消息后，需要遍历链表，才能找到需要的消息，然后将其插入到链表的首部。其平均时间复杂度为 $O(n)$ 。最好情况下第一个就是需要的数组元素，最坏情况下需要遍历整个链表才能找到。其次，在移动的过程中仅需要调整指针调整的时间复杂度为 $O(1)$ 。整体的空间复杂度为 $O(n)$ ，即整个链表。

使用数组时，可直接使用数组下标访问，由于消息由时间戳为标准有序排列，额外可以做到在查找某个时间段消息时可以使用二分搜索从而达到 $O(\log n)$ 的平均时间复杂度。但是在移动元素的过程中需要大量进行元素的前移（消息只会往数组后端插入或移动）。其将产生平均 $O(n)$ 的时间复杂度，所以使用数组的平均时间复杂度仍为 $O(n)$ ，空间复杂度为 $O(n)$ 。

2.1.3 使用 Map 实现

常用的 Map 分为哈希表和 B-树系列(本文采取 B*树)。当采用哈希表时，虽然可以将元素查找及插入删除的平均时间复杂度都将下降至 $O(1)$ 却失去了可迭代的能力，且不能根据数组下标直接访问元素。

使用 B*树时，可以根据元素的时间戳和下标进行双索引。假设 B*树为 m 阶 B*树(m 一般大于等于 3) (B*树的具体定义请参考其他教材)。B-树在本文中很重要的特征是，下层结点内的值总是落在由上层结点值所划分的区间内。所以，我们可以以时间戳为主要索引构建整颗 B+树，如图 2-1，`data` 指针指向数据地址。再将所有数据由时间戳从大到小链接，时间戳最大者为首。这就实现了能够顺序迭代，且能够快速查找位置进行修改操作，此时一个的 B*树已经按照需求构建。

其次，为了实现根据下标查找，每个 B*树节点应附带右侧子节点的数量。如下图 2-1，50 节点应包含 `Bigger=4`。在进行下标查找时，假设当前查找下标为 n 的节点(n 应小于节点数减一，且大于等于 0)。若 n 等于 `bigger+1` 则当前节点即为要查找的节点；若 n 大于 `bigger+1` 则说明节点在当前节点的左侧，小于则在右

侧。注意：在向右查找时 n 不需操作，但当向左查找时， n 需要减去右侧节点的数目。

如查找 $n=6$ 的节点，首先进入 50，其 $Bigger=4$ ，判断需要向左滑动， n 修改为 2。此时根据 50 指向的 26 进行检索，其 $Bigger=1$ ，26 即为我们需要的节点。

如查找 $n=3$ 的节点，首先进入 50，其 $Bigger=4$ ，判断需要向右滑动， n 不变。此时 75 节点的 $Bigger$ 为 3，不满足，向右滑动到 80，80 的 $Bigger$ 为 1，向左滑动到 76， n 修改为 1，76 的 $Bigger$ 为 0，此时 76 即为需要的节点。

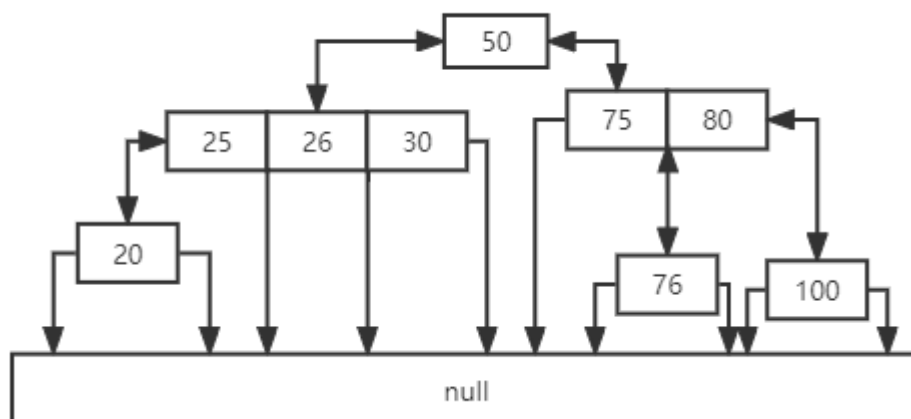


图 2-1 二阶 B*树实现示意图(时间戳从大到小进行串联未画出)

B*树的实现在时间上，根据下标查找和根据时间戳查找，从根节点到关键字所在节点的路径所涉及的节点数不超过：^[1]

$$\log_{\lfloor m/2 \rfloor} \frac{N+1}{2} + 1, \text{ 其中 } m \text{ 为阶树, } N \text{ 为关键字个数}$$

所有的删除操作中，向上关系凡是父-右子树关系的父节点都需要减 1，遇到其他关系结束。

如果只有一个值的节点被删除，被删除节点的右子树的最左节点代替，如果没有右树直接将左树顶替当前节点的位置。如果没有子树，直接删除即可，这些情况下 $Bigger$ 都不要调整。如果删除值是某个节点的非最左节点，其左侧节点的 $Bigger$ 值都需要减 1 处理。

如将 20 节点删除，向上检索，没有父-右子树关系的父节点，执行 B*树的删除逻辑。如果删除 75 号节点，则 50 节点的 $Bigger$ 需减 1，再向上没有父-右子树关系的父的节点，然后执行 B*树的删除逻辑。如果删除 100，向上分别需要修改 80、75、50，最后执行 B*树的删除逻辑。

如果加入节点，必定是在最右侧加入，此时向上检索其父-右子树关系的父节点都需要将 $Bigger$ 加 1。

最后是触发 B*树调整操作。分以下几种情况：

¹ 严蔚敏,吴伟民.数据结构(C语言版),清华大学出版社,2007 第 240 页 B-树查找分析

当前插入节点需要分裂时，分裂后父节点、当前节点、新的右子树的 **Bigger** 不变，新的左子树的 **Bigger** 都将重新计算，其值为到新左子树最右值的距离加最右值的 **Bigger** (本设计中这个最右值的 **Bigger** 必为 0)，如图 2-1 中 26 节点将分裂为新的节点 25 节点值变为 0，26、30 节点不变。

当需要合并时，右侧不变，左侧合并来的节点将加上原先节点最左侧的值的 **Bigger**，右侧合并上来的值和其他节点都不变。

其缺陷是，如果不及时压缩树的高度，其将变成一颗只向右拓展的树。所以还需经常调整树的结构。方案一是当插入节点时，如果根节点右侧节点数量大于左侧（左侧数量由总的大小减去 **Bigger**），需要把右侧节点上提（类似于二叉平衡树的平衡操作）。方案二是插入时向上合并上层节点，以触发的分裂来重构较为平衡的树。

以方案一为例，当最右侧插入节点导致分裂影响到左右高度差不可接受时。对树从根节点进行左旋，由于 **B*** 树的性质，父节点的值一定是小于右子树的所有子节点和子树，故原先的右节点的左节点最左值一定大于根节点的最右值，所以右节点的最左节点将成为根节点的新右节点，而根节点将成为新根节点的左节点。同理，右旋操作中，需要将最左节点的最右节点(包括空指针)，作为根节点的新最左节点，原先的最左节点的最右节点为原根节点。其 **Bigger** 的调整类似上方，某个节点如果被左旋上来，则无需调整，但被左旋下去的节点需要重新计算 **Bigger** 值其所有左子节点不变，右旋同理。简而言之，“上不变下变”。

此时如果不继续对两边子树进行检查操作，则可能得到一棵两边粗壮中间稀疏的二叉树（只有两个枝）。调整完后的根节点需要检查各自的左右节点，其左节点只会进行右旋，其右节点指挥左旋。

由于本人材浅学疏，无法证明本方法的时间复杂度，只能大概猜测时间复杂度大致在 $O(\log_m n)$ 左右。

2.1.4 LRU 算法的实现

LRU 算法是最主要的页面置换算法之一，其基本思想在于，通过快表存储最近使用的页面，淘汰最旧不使用的页面。其思想非常契合消息盒的需求。

在算法实现上，首先所有的消息都将以双向链表的形式进行存储，构造一个新消息在上，旧消息在下的栈的结构。其次，记录一个“到栈底的距离(下文称距离)映射到消息地址”的哈希表。当有新消息时直接压入栈顶即可，新消息到来或发送消息时，只需要将消息上层所有的节点的距离减 1 即可。由于用户聊天的时间局部性(经常聊天的对象一定经常发消息，已经聊起来的用户之后也可能继续聊)，所有不经常聊天的一定会被下沉到栈底，经常聊天的一定会浮动在栈的上方，所以在栈顶修改索引代价一定是一个较小的代价。

在使用索引随机访问时，第一个元素即为时间最晚的元素，即栈顶元素，第二个为栈顶向下的第二个元素。索引的转换为：栈的大小-索引-1。

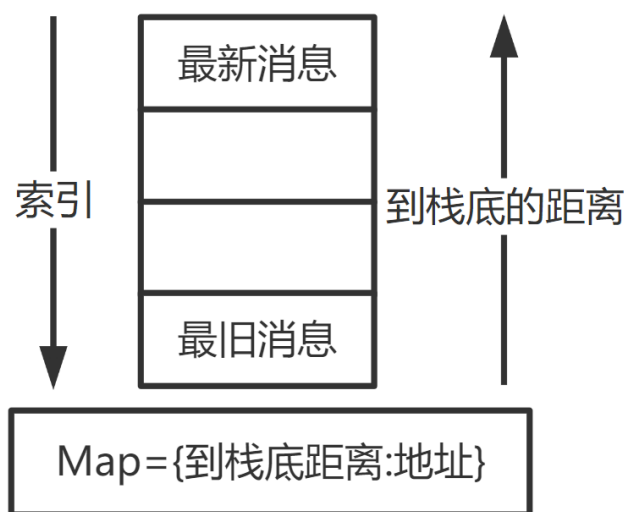


图 2-2 LRU 实现示意图

在 LRU 的设计中，查找的操作毫无疑问是 $O(1)$ ，但是由于用户操作行为难以统计，修改的平均复杂度无法计算，但若以最坏情况下均匀分布作为用户操作，LRU 设计仍有 $O(n)$ 的最坏时间复杂度。在空间复杂度上，首先双向链表需要记录前后的索引，其次哈希表需要记录距离和一个指针，平均每个数据需要三个指针一个数字平均空间复杂度为 $O(n)$ 。其优化点在于：不经常聊天的对象不加载入内存，这不仅优化了加载速度，也减少了内存开销，把内存用到刀刃上。在实际使用中，LRU 算法以其简单实现，性能也不弱，故被各大软件广泛使用，同时也是大厂校招热门算法之一，其替换策略也是基于非常经典的局部性原理，非常值得深入探讨。

2.2 自定义网络协议

2.2.1 总协议

如图 2-3，Valid 字段用于描述上一个报文的有效性，如用户发起登录，服务器检查后如果登录成功 valid 字段会置 1。其他消息中，如果用户发送的消息不合规，服务器将返回 valid=False 的协议体告知请求失败，具体错误信息见 pairs

ConnectType 用于记录连接的种类，如请求响应和中转，请求是向对方请求数据，具体需要什么需要在 config 设置。响应则为响应对方行为。中转主要是服务器中转各个用户之间的操作。

Millisecond_timestamp 记录了毫秒时间戳，是非常重要的报文有效性评估内容。

映射 Config 存储了绝大多数设置，如客户端发送方的 UID，请求方法，数据类型等。

映射 Pairs 存储键值对信息，主要是零散消息的存储。以及数据的编号等。

映射 Bits 是主要的存储手段，将所有数据、图片等二进制形式的数据进行传送，同时所有子协议都将直接变为二进制字符串传送，用户根据需要反序列化。

```
message Protocol
{
    bool valid = 1 ;

    ConnectType type = 2;

    uint64 millisecond_timestamp = 3;

    map<string, string> config = 4;

    map<string, string> pairs = 10;

    map<string, bytes> bits = 11;
}

// 连接种类
enum ConnectType
{
    // 请求
    REQUEST = 0;
    // 响应
    RESPONSE = 1;
    // 中转
    TRANSIT = 2;
}
```

图 2-3 总体通讯协议

2.2.2 用户消息

用户消息是单独描述用户之间的协议，其主要囊括了消息的类型、消息内容、时间戳、来源和去向。消息类型描述消息内容的属性，如果是文本则消息内容直接装文本，如果是图片，则消息内容存储图片的 URI，具体的文件在 Bits 中以 URI 映射 bytes 获取。来源去向时服务器用来转发消息的依据，没有这个数据，服务器将无法分辨数据的去向。

2.3 客户端其他数据结构

2.3.1 返回结果

返回结果是一个封装类，封装一个泛型，代码如下：

```
Ssealed class Result<out T:Any> {
    data class Success<out T:Any>(val data:T): Result<T>()
```

```
data class Error(val exception:Exception): Result<Nothing>()
}
```

代码 2-1 Result 类的定义

可见，上层只需要调用反射查看是什么类即可判断成功还是失败，如果成功，Result 将自带一个 data 的数据。该数据结构封装了所有的结果，上层可以方便的知道下层发生了什么错误，返回了什么值。摒弃了返回值或抛出异常的处理方式，同时也方便反馈用户发生了什么错误。

2.3.2 联系人列表

联系人列表需要根据 UID 获取联系人，使用下标来进行访问。所以联系人列表只需一个 UID 映射地址的 Map，和数组即可实现。Map 可以选择 HashMap 或 B 树，如果选择 HashMap 将提供 O(1) 的访问平均时间复杂度，如果使用 B 树，可以做到模糊搜索，为用户提供搜索功能。

2.3.3 联系人

联系人数据结构需要存储联系人的基本信息，即 UID、昵称、图标、显示名、标签和联系人类型。

2.3.4 用户数据

由于运行时需要一些必要数据来辅助应用运行，故存储状态、UID、昵称。其次聊天数据、联系人列表都需要动态存储。

2.3.5 安全数据

考虑到数据传输的安全性问题，根据 PGP 的基本原理。用户会在最开始与服务器构建连接时构造钥匙串。钥匙串首先包括一个用户的私钥和公钥，其次是服务器的公钥，最后是会话密钥、时间戳等会话级安全数据。在数据传输时，用户首先用自己的密钥进行加密，然后对报文进行哈希处理，然后将加密报文用服务器公钥进行加密，发送至服务器，服务器可以使用自己的私钥进行解密，然后校验报文完整性，用客户端的公钥解密。然后服务器将随机生成一个会话密钥，以相反的方式发送至客户端。此后双方以会话密钥进行加密报文。本设计中预留了这个数据位，但由于本人精力原因无法实现数据安全方面。故设计中以 encoding 代替加密。

第三章 客户端设计

3.1 UI 设计

3.1.1 总体设计

UI 总体采用主 Activity 通过 Fragment 显示消息、联系人、“我的”三个界面。Fragment 存在复用的地方，如消息、联系人、“我的”都有一个标题栏一个 RecyclerView 组件。故抽象出抽象类 BaseFragment 在 onResume 后提供一个 onInit 方法，供子类进行组件绑定等。

其次，由于网络通讯模块使用多线程异步执行，而 RecyclerView 的更新需要调用 RecyclerView 实例的 adaptor 的 notifyDataSetChanged()方法，故 BaseFragment 需要在 Fragment 构建时构造为单例模式，并实现观察者模式，以便网络通信成功和对本地数据修改后能及时唤醒 UI 更新显示内容。另一方面，也可以实现定时调用方法或者在 onResume 拉取。

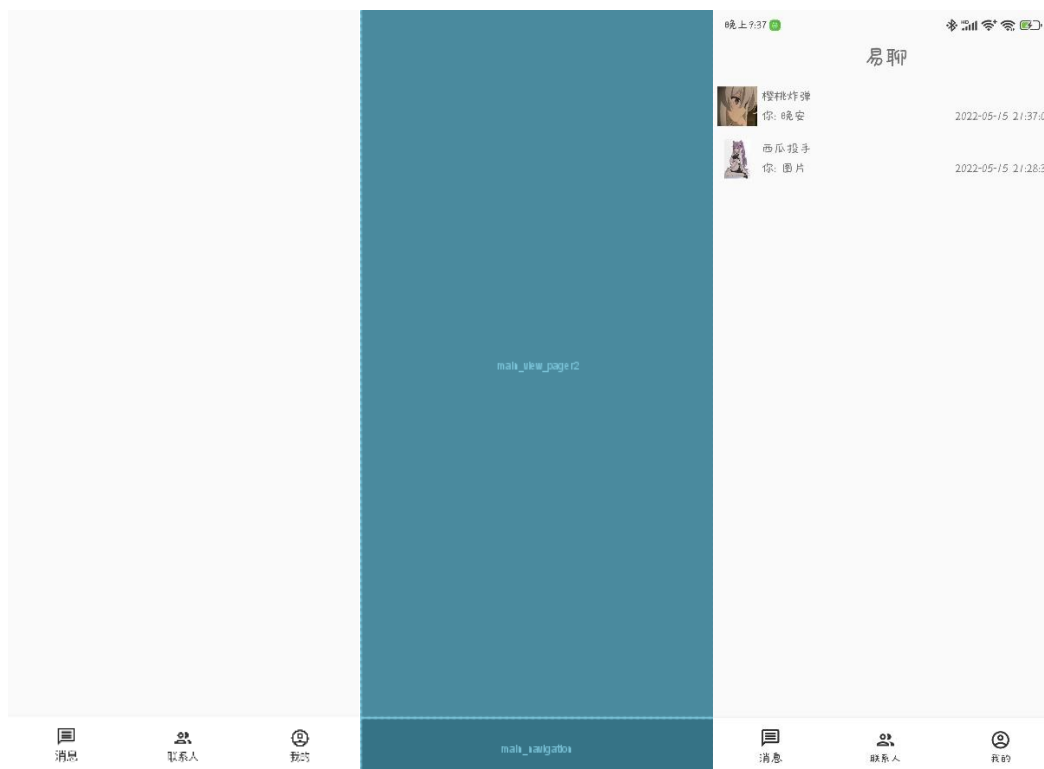


图 3-1 用户主要界面(左)、布局设置(中)、实际效果图(右)

3.1.2 消息界面设计

消息界面需要能够下拉刷新，上拉加载，故需要定义一个 MessageFragment 实现 BaseFragment，其布局文件和 Activity 应包含 SwipeRefreshLayout 及其初始化内容。根据需求，新的消息必须在消息栈的顶端，所以 RecyclerView 的 adaptor 的数据应指向 2.1 定义的消息盒子，本设计具体实现了 2.1.4 的 LRU 实现。

3.1.2 联系人界面设计

联系人界面同样需要下拉刷新，上拉加载。在微信的设计中，用户被排序、按拼音序划分并有 sidebar 辅助跳转，本设计将实现排序，按自定义规则进行划分。如系统联系人、按自定义规则排序等。

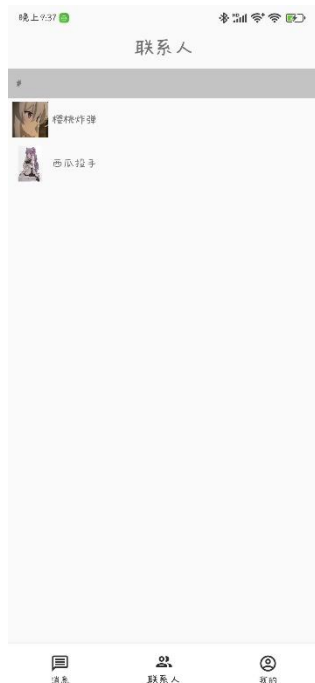


图 3-2 联系人界面效果

3.1.3 “我的”界面设计

“我的”界面主要囊括系统设置等，如退出登录等一系列操作。



图 3-3 “我的”界面效果

3.1.4 聊天界面设计

当用户点击消息或联系人时，需要跳转到聊天界面，聊天界面应包括输入框、功能键、菜单栏、双方消息、标题栏、返回键等。

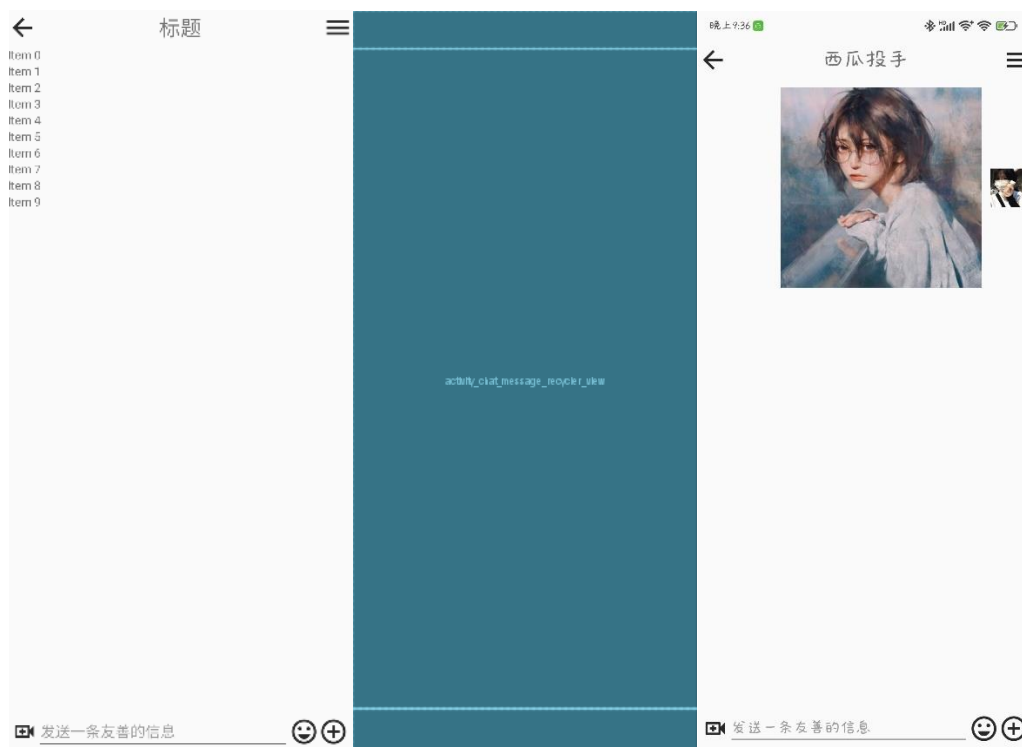


图 3-4 聊天界面(左)、设计(中)、实际表现(右)

聊天的消息需要独立设计通用的布局，以兼容不同的消息类型。可以抽象基础的布局为一个抽象类。基础的布局应该包括用户的昵称、头像。其继承类为发送和接收，即当用户接收消息时，对方图片昵称文字应左对齐。当用户发送信息时，其昵称头像应右对齐。再次继承则构造出针对不同消息类型的布局。如图片应包含一个 `ImageView`，文字应包含 `TextView`，定位信息则视情况调整。

3.2 通讯设计

3.2.1 Socket 封装类设计

用户聊天数据需要确保消息的可靠性，所以需要选择使用 `TCP` 进行传送。本设计中客户端主要有两种与服务器沟通的方式。其一是客户端向服务器发送请求、并期待得到响应。这种“`One request one response`”的短连接模式具有非常广泛的使用。短连接对于服务器来说实现较为简单，实时存在的连接都是有用的连接，不需要额外的控制，但如果客户端频繁连接中断，则在 `TCP` 的建立和关闭上会造成时间的较大的浪费。对于聊天软件来说，用户并不是每时每刻都在发送消息接收消息，对消息的实时性没有网络游戏那么高。所以完全可以采取短连接的

方式。用户第二种和服务端交换数据的方式是监听端口，其主要的用途是当另一用户发送了消息，服务器需要通知用户取走数据来避免服务器资源的浪费。

为什么要封装 `socket`？不管是 `socket` 通信程序的客户端还是服务端，其代码都是又长又难看，如果不加以管理将极大影响主程序的结构，必须分离出来。当项目越来越大，对 `socket` 资源的管理也越发重要，如果存在不需要且没有及时关闭的连接，不管对于服务器还是客户端都是比较严重的资源浪费。

在封装的过程中，需要注意把数据初始化的代码放在构造中。在 C++ 中把关闭 `socket` 等释放资源的代码放在析构函数，Java 在 `finally`，Python 则对应 `__del__` 中。在类声明时，应当注意把 `socket` 定义为类的成员变量，类外部的代码根本看不到 `socket`，确保 `socket` 完全封装。同时，由于 `socket` 仅是操作系统对 TCP/UDP 的简单封装，如果没有进一步封装，需要程序员对 TCP/UDP 具有一定的了解才能熟练编程。封装后的 `socket` 能做到代码简洁、安全、自动释放、使用简单。

3.2.2 协议构造器

建造者模式是开发过程中非常常用的模式之一，它常用于创建过程稳定，但配置多变的对象。比如本设计的总协议中有非常多的协议种类设置、参数设置。如果每次都手动调用，代码将会非常复杂，不利于开发及维护。其次，经典的“建造者-指挥者”模式已经不太常用了，其原因是指挥者的作用是隔离用户与建造过程，但这不利于调用且与工厂模式有一定重叠。现在建造者模式主要用来通过链式构造调用生成不同的配置。如当前需要构造一个协议，除了必要的配置信息外，还需要知道协议的种类划分，是图片、文本还是其他。

代码 3-1 所示：用户的请求较为复杂，我们可以将协议的构造方法设置为私有。外部不能通过 `new` 来构建协议实例。对于必须的属性，可以在 `build` 方法的参数中强制给出，或生成默认值。可选属性可以通过构造器模式的链式调用方法传入，上层可以根据不同的配置构造出需要的协议体。

```
class ProtocBuilder{
    init;

    fun buildValid(): ProtocolOuterClass.Protocol;
    fun buildInvalid(): ProtocolOuterClass.Protocol;

    fun putBytes(key:String,value:ByteString): ProtocBuilder;
    fun putsBytes(map:Map<String,ByteString>): ProtocBuilder;
    fun clearBytes():ProtocBuilder;

    fun putPairs(key:String,value:String): ProtocBuilder;
    fun putPairs(map:Map<String,String>): ProtocBuilder;
    fun clearPairs(): ProtocBuilder;
```

```

fun requireLogin(uname:String,upassword:String): ProtocBuilder;
fun requireRegister(uname:String,upassword:String): ProtocBuilder;
fun requireLogout(): ProtocBuilder;
fun requireMessage(): ProtocBuilder;
fun requireContacts(): ProtocBuilder;
fun requireContact(des_uid:String): ProtocBuilder;
fun requireImage(names:List<String>): ProtocBuilder;

fun sendMessage(des_uid:String,data:UserMessage): ProtocBuilder;
fun responsePositive();
}

```

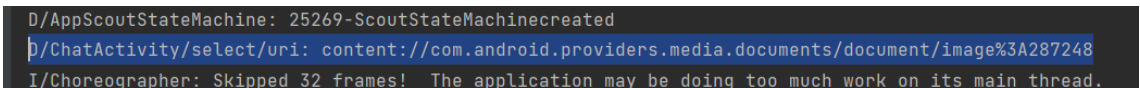
代码 3-1 协议的建造者模式接口展示(具体方法见代码)

3.3 资源管理系统

3.3.1 本地数据管理

本地数据库主要用于存储资源地址，消息记录。消息记录主要包括四个字段：对方的 UID、消息内容、时间戳、消息种类。如图片等资源种类为图片，消息内容将存储其路径或 URI（统一资源标识符，是一个用于标识某一互联网资源名称的字符串）进行唯一标识，时间戳是时间的标识。数据库可以设计为两种模式：所有用户公用一个表和每个用户一个表的方式。前者对构造和操作更为简单，后者的性能更佳。

文件存储是 Android 中最基本的一种数据存储方式。但是由于安卓 Q 以后对用户隐私的设定，开发者不能直接访问外部文件。所以这里会间接通过使用 ContentProvider 获取资源的 URI 进行访问。每个 ContentProvider 都会对外提供一个公共的 URI，如图片在 Content 通常表示为“image/*”，当调用 toString()后的结果如下图 3-3。如果，应用程序需要进行数据的共享时，要使用 ContentProvider 为这些数据定义一个 URI，然后其他的应用程序可以通过 ContentProvider 传入 URI 来对数据进行操作。



```

D/AppScoutStateMachine: 25269-ScoutStateMachinecreated
D/ChatActivity/select/uri: content://com.android.providers.media.documents/document/image%3A287248
I/Choreographer: Skipped 32 frames! The application may be doing too much work on its main thread.

```

图 3-3 某图片的 URI 调用 toString()方法后的输出

SharedPreferences 是常用的键值对的方式的轻量级的数据存取方式。而且还支持多种不同的数据类型存储，读取出来的数据必定与其存入的数据类型相同。在本实现中，SharedPreferences 被用来保存用户的登录数据以方便下一次打开后能直接自动登录。使用 SharedPreferences 的第一种方法是在 Context 类中开发者手动调用 getSharedPreferences()，用户可以指定访问权限和名字，如果文件不存

在则会自动新建。而第二种 Activity 类中的 `getPreferences()` 方法只接收一个操作模式参数，自动将当前活动的类名作为保存的 `SharedPreferences` 文件名。第三种方法是调用 `PreferenceManager` 类中的 `getDefaultSharedPreferences()` 静态方法，它可以自动使用当前应用程序的包名作为前缀来命名 `SharedPreferences` 文件，在获取到 `SharedPreferences` 文件对象后，向其写入数据。

最后是使用 Android 系统内置数据库 `SQLite`。`SQLite` 是一款轻量级嵌入式数据库引擎，其不仅支持 `SQL`，还遵循 `ACID` 事务。其专门提供了一个帮助类：`SQLiteOpenHelper`。在本设计中，`SQLite` 可以用于聊天数据的持久化存储。原因是 `SQLite` 可以利用很少的内存就有很好的性能。而且聊天数据并不是一直都要看。

3.4 业务逻辑设计

3.4.1 登录注册逻辑

客户端的登录注册逻辑为用户输入好数据，进入文本检查。首先需要检查用户名和密码的位数，过低的密码位数并不安全。其次考虑到用户密码的安全性，服务器存储密码往往是存储密码哈希值，放置服务器被攻破时可能造成的用户密码泄露。其次，由于传输过程使用明文通讯，服务器容易受到重放攻击，理应使用 `PGP` 加密或 `SSH` 使用会话密钥对重要会话信息进行加密。然后构造协议体，向服务器发送协议，等待服务器的响应。服务器的响应将在下个章节进行阐述。当客户端接收服务器信息时，会向上返回 `Result.Success` 或 `Result.Error` (见 2.3.1) 上层可以直接判断 `Result` 是否是 `Success`。如果是，可以直接调用 `Result.data` 获取数据，如果不是，可以直接调用 `toString()` 方法获取错误信息。

3.4.2 联系人逻辑

联系人逻辑分为两类，一种是查询另一种是修改。前者比较简单，当用户构造协议体提出查询请求时，服务器将所有用户信息返回至用户，由客户端检查当前显示的联系人列表是否需要更新或替换。需加载其他内容，如头像，将采用懒加载的模式，即只有当用户到这个界面了，`RecyclerView` 执行到 `onBindView`，图片才会进行加载。客户端根据头像的 `URI` 会对本地资源进行检查，如果找不到，会开启一个新的线程向服务器请求数据，不管返回结果如何客户端都会重新根据 `URI` 再次检查本地文件，如果存在即为拉取成功，不存在则代表着服务器也不存在在该文件或连接出错，客户端则显示默认图片。

3.4.3 消息机制

客户端接收消息需要主动或被动向服务器查询消息。两者实际上都是客户端请求的方式，区别是前者由客户端定时触发，后者由服务器尝试触发。前者需要实现简单的 `RPC` (`Remote Procedure Call`，远程过程调用)，后者保持一个全局计

时器和 Event 触发机制即可。客户端请求数据后，服务器会返回对应一个协议体，协议体的 bits 段将全部用于消息内容的存放。客户端根据服务器返回的数据检查是否需要更新 MessageBox 和 Message 内容。

客户端发送消息就较为简单，使用对应的消息协议构造器构造消息，然后通过封装的 Socket 直接发送即可。

3.5 锁机制

客户端除了 UI 操作，其他绝大多数操作都使用了多线程操作，其次用户本地有非常多的静态数据例如联系人、消息、登录状态信息等。线程安全问题便由这些全局变量和静态变量引起，上述绝大多数信息都是全局静态变量包括但没有提到 socket 资源。多个线程同时对同一对象进行写操作，就可能发生每个线程读取到的值不一样。对于 socket 来说就是对服务器重复发送消息或脏消息。在 Kotlin 中线程锁可以分为实例锁和全局锁。实例锁通过注解 `@Synchronized`，其本质上是使用 Java 的 `Synchronized` 关键字。其基本规则是：对于普通方法，锁的是当前对象、对于静态方法，锁的是 `class` 类、对于代码块加锁，锁的是代码块匿名对象。Java 的 `Synchronized` 关键字可以保证加锁后的各个对象在多线程环境下只在一个线程能够执行，即在当前线程释放对象锁之后其他线程才能获取执行。并且 `Synchronized` 是阻塞的，如果首先获取锁的对象、线程没有正确退出或一直占用不释放，可能会导致线程锁死。在 Kotlin 中 `@Synchronized` 注释的调用方式使用如下：

```
@Synchronized
fun tick(){ ... }
```

代码 3-2 `@Synchronized` 锁方法使用示例

在某些情况下某个类拥有多个对象，运行过程中对象本身需要被锁住，则需要使用 `synchronized()` 来锁定类的实例。这样多个对象之间不会互相影响，A 对象锁死不会影响 B 对象的访问。

```
Thread{
    synchronized(类实例){ ... }
}.apply{
    start()
}
```

代码 3-3 锁类代码使用示例

如果 `synchronized` 的参数是类，即 `ClassName::class.java`，那所有的对象都会共享同一个锁。

锁的另一种实现是 `lock` 接口，而 `ReentrantLock` 是 Kotlin 内置的可重入锁。其使用方法为 `lock.lock()` 和 `lock.unlock()`。`lock()` 获取锁，获取成功将会设置当前线程的 `count++`，如果其他线程占有锁，则当前线程不可用，进入阻塞。`tryLock()` 会尝试获取锁，如果锁不可用，则此方法将立即返回，线程不阻塞，可设置等待时间。`unlock()` 则尝试释放锁，会使当前线程的 `count--`，如果线程的 `count` 为 0，则释放锁。

在 Kotlin 的协程中提供了 `Mutex` 来保证互斥，对于多个协程来说用的是同一个 `Mutex`。直接使用 `mutex.withLock {}` 就可以使用。要注意的是，在协程中，`@Synchronized` 不起加锁的作用，即无效。

锁的使用非常广泛，比如本设计中的消息列表、联系人列表都是非常典型的单例模式。其次，使用短连接时当用户向服务器请求数据，用户只能进行唯一的请求，否则不仅浪费了服务器的资源，而且当服务器返回的多个数据到达时，将不可避免的产生时序问题。

3.6 消息的传递

文本消息：文本消息客户端直接调用协议构造器，构造出来源、去向、内容、文本类型即可直接向服务器发送消息。

字节流消息：所有的图片等文件都将被是做字节流。当传输图片时，客户端先进行 `hash` 处获得图片的唯一标识（哈希碰撞的概率较低，可忽略不计）处理后对图片进行 `Base64` 编码，编码后同文本消息构造来源、去向、内容、图片类型即可发送消息。在接收消息时，当客户端根据 `hash` 加载图片，会检查本地有没有这张图，如果没有，将请求服务资源。接收到资源后自行 `Base64` 解码并保存。然后显示图片。

第四章 服务端设计

4.1 通讯设计

4.1.1 Socket 监听器

服务器需要持续监听各客户端的数据，并且响应数据。需要专门设计一个监听类，方便管理，监听器需要具备监听信息、转发信息的功能。

服务器接收消息有四种方式：

(1)阻塞，即调用函数没有接收完数据或者没有得到返回值之前，不会返回。线程或进程阻塞等待结果。

(2)非阻塞，即调用函数立即返回，通过 `select` 通知调用者，其有三个返回值分别是可读可写和异常。在 Windows 和 OpenVMS 中 `select` 只支持 Socket，在 Linux、Unix 所有的文件系统也支持。

(3)同步，即调用一个功能，该模块执行没有结束前，阻塞等待结果。

(4)异步，即函数调用一个模块，且在该模块有结果或有返回值后通过回调通知主线程。

本设计采用了阻塞的方式。在阻塞方式下，客户端 `connect` 首先发送 SYN 请求到服务器，当客户端收到服务器返回的 SYN 的确认时，`connect` 返回，否则的话一直阻塞，直到触发超时。其次，服务器在接收到信息之前，将一直保持阻塞状态。而一旦有堵塞，则表示服务器与客户端获得了连接。所以本设计的主线程接受连接请求，使用其他线程处理与客户端的连接。其次，阻塞模式下调用 `accept()` 函数，在没有新连接时，线程会进入睡眠状态，直到有可用的连接进程线程会被唤醒。

在阻塞和非阻塞中，调用 `Socket.recv()` 的返回值没有区分，都是小于 0 表示出错，等于 0 表示连接关闭，大于 0 表示接收到数据大小。两者区别是：阻塞时调用 `read()` 和 `recv()` 等 socket 函数时会一直阻塞，直到有数据才返回。而非阻塞不管接收缓冲区中有没有数据，函数调用都离开返回。

阻塞的 socket 存在一系列问题：`connect/accept/write` 会导致线程阻塞，即使是服务关闭、重启线程也无法退出；`recv` 读数据长度不确定，这导致不能保证数据的完整性，最终可能导致无法解析协议的问题。可以通过先传输数据大小（需要客户端、服务端提前约定数据大小这个数字的大小）然后再传输数据。

相反，非阻塞的 socket 主要存在技术实现的问题：在建立连接的时候要兼容处理“返回 `EINPROGRESS`”的情况，在 `connect`、`read/recv`、`write/send` 时要兼容处理“返回 `EWOULDBLOCK`”的情况。在非阻塞 socket 的实现中，建议自行将 `read()` 和 `recv()` 封装，并且在超过一定时间内进行心跳，收发不了数据就算做异常。否则会出现资源浪费的情况。

本设计的阻塞 socket 通过 IO 复用模型、多线程、集中调度实现 socket 阻塞的调用。其次，在报文格式方面，通过短连接和使用 Base64 对报文进行编码约束报文内容，然后在报文结束时，发送二进制的“#”，来解决阻塞 socket 长度不确定的问题。服务器执行读操作直到出现“#”就立即进入解析等后续操作。

4.1.2 Socket 发送器

当服务器接收到用户发来的信息，需要对其进行转发，转发主要分为两类。第一类是普通消息的转发，第二类是联系人类型的转发。而普通消息的转发又分为 Unicode 文本和二进制文件的转发。使用的方法仍是上文的 Base64 编码和短连接。

在 Socket 写操作 write() 和 send() 中，返回值和读操作一样没有区分，都是小于 0 标识出错，等于 0 表示连接关闭，大于 0 表示接收到数据大小。阻塞模式下，如果发送缓冲区没有空间或者空间不足的话，write()、send() 会阻塞，如果有足够空间，则拷贝所有数据到发送缓冲区，然后返回。这要求客户端不能出现阻塞情况，客户端也是使用多线程来解决这个问题。

4.2 数据库设计

用户表：用户表分为用户唯一数据表和用户基础数据表。唯一数据包括 UID(用户唯一标识 ID，主键，唯一，自增)，uname(用户账号，超键，唯一)，upassword(密码，唯一)。用户基础数据包括了用户所有能自主设置的值，包括昵称、头像编号、id 等。

用户关系表：用户关系表存放用户之间的关系，主要字段为 src_uid、tar_uid、nickname、relationship 等。一条记录代表着谁对谁以什么称呼为什么关系。

4.3 调度器设计

4.3.1 主调度器

调度器控制着何时何主体调用何方法，它知道如何根据优先级或其他标准来存储任务和将任务进行排序。调度器为各种消息和请求提供了中控管理能力，从而可以重用代码减少重复。在 Java 中调度器是一个异步执行框架。它基于生产者-消费者模式，任务的生产者(发送)与任务的消费者(执行)是分离的，是不同的线程。在大型项目中，往往是使用“调度器-服务器-文件系统”的三级模式。由于实现难度和个人经历原因，本设计采用的是“服务器-线程-文件系统”的模式，调度器主要用于分配任务、分配线程资源。

4.4 服务器交互逻辑

4.4.1 登陆注册

用户向服务器发起请求后，服务器 **Socket** 监听器接收到连接请求。然后调度到其他线程，本线程进入阻塞状态，直到客户端发送到二进制的“#”。线程拿到接收的数据后，对其进行解码。判断任务类型为登录，调度器将任务分配线程，进入 **logic** 层，**logic** 层主要负责逻辑判断，返回协议构造。经由 **logic** 层检查后如果用户不符合标准将拒绝后续的数据库访问。反之，逻辑进入 **db** 层。**db** 层负责数据库的访问逻辑。如果用户登陆数据没有问题，会返回用户的 **UID**，逻辑层则会构造返回数据集，其包括 **UID**、用户名、昵称、头像、**key** 等。注意：此时头像不包括其二进制文件，只是文件编号。

4.4.2 转发逻辑

当用户发送信息且服务器正常接收后，服务首先检查是否含有图片等二进制文件，如果存在，则直接保存为二进制文件，不对其进行 **Base64** 解码。这么做的理由是，服务器不需要知道用户发送了什么，所有的编码解码工作都由客户端自己进行。在运行时，服务器保存一个消息池和 **IP** 池，用于缓存所有的用户消息。每当服务器接收到消息，会尝试从 **IP** 池联系用户，如果用户在线，且在线用户是目标用户，服务器会向客户端发起提醒告知用户你有新的消息，请去拉去。其次客户端还有定时拉取的机制。两者的服务器都不直接发送消息，其主要原因是不确定用户的网络状态，由于使用了短连接的方式进行通讯，消息如果不是客户端主动拉取，会产生较大的时延。

第五章 设计评估

5.1 功能评估

5.1.1 测试用例

系统评估需要设计测试用例，本次的测试用例如下：

表 6-1 功能测试用例

| 模块 | 场景/目标 | 描述 |
|--------|----------------------------|--------------------------------------|
| 网络功能 | 网络异常（弱网） | |
| 登录 | 输入正确的用户名和密码， 点击提交按钮 | 验证能正确登录 |
| 登录 | 输入数据库不存在的用户名 和密码，点击提交按钮 | |
| 注册 | 输入已经存在的用户名 | 不能注册并提示重名 |
| 登录/注册 | 什么都不输入，点击提交按钮， 看提示信息。 | 提示用户输入 |
| 登录/注册 | 输入错误的用户名或者密码 | 提示用户错误 |
| 登录/注册 | 用户名或密码输入 5 位 | 提示用户长度不足 |
| 登陆界面 | 旧手机打开 | 低于安卓 Q 不能打开 |
| 登录界面 | 全面屏手机观察界面 | 界面美感，颜色搭配合理， 文字的大小和颜色搭配合理 |
| 登录界面 | 平板电脑观察界面 | 界面不畸变，输入框不遮挡 |
| 登录 | 登录成功花费时间(广域网) | 统计登录的花费时间 |
| 登录 | 登录成功花费时间(局域网) | 统计登录的花费时间 |
| 登出 | 登出能否释放账号独有资源 | |
| 登出 | 检查自动登录是否失效 | |
| 登出 | 检查服务器资源是否释放 | |
| 记住密码功能 | 用户首次登录后再次打开直接 登录 | 以上次登录用户为准自动登录 |
| 记住密码功能 | 登录失败后再次打开不能直接 登录 | |
| 昵称设置 | 用户设置昵称 | |
| 昵称设置 | 空输入框设置昵称 | |
| 头像设置 | 用户设置头像 | |
| 头像设置 | 取消设置头像 | |
| 密码设置 | 用户修改密码 | |
| 密码设置 | 密码输入不合规 | |
| 文本消息发送 | 发送文本信息 | 兼容所有 Unicode 信息 |
| 文本消息发送 | 发送 Emoji 信息 | 兼容所有 Unicode 信息 |
| 文本消息发送 | 文本 Emoji 混发 | 按正常序编码解码 |
| 图片消息发送 | 发送图片信息 | 测试返回成功的 registerForActivityResult |
| 图片消息发送 | 取消发送文件 | 兼容返回失败的 |

| | | |
|-------|----------------------|---------------------------|
| | | registerForActivityResult |
| 消息拉取 | MessageFragment 下拉拉取 | |
| 消息拉取 | 用户登录后自动拉取 | |
| 联系人拉取 | ContactFragment 下拉拉取 | |
| 联系人拉取 | 用户登录后自动拉取 | |
| 添加联系人 | 输入了用户名添加 | |

结束语

此次毕业设计是我耗时最长的项目之一，其设计、编码、测试整个流程的完整性令本人收获颇多。其次，从需求搜集、代码编写、白盒测试。我了解到如何在软件开发的实际过程中面对“正确的理解和管理需求及其变更”的问题。同时，此次毕业设计是我完全自己构思架构再逐步实现的过程，这全面锻炼了我四年不同的专业知识，是我对四年知识进行了系统性、实践性的复习。

由于时间紧迫，客户端的本地存储模块并未实现，服务器端的消息缓存机制也堪堪只用一个字典解决。由于本人知识有限、材浅学疏、时间不足，论文不可避免会存在一些疏漏之处，在此后的工作中我会精益求精，不断改进。

致 谢

首先非常感谢我的两位指导老师还有同学们，他们在论文写作过程中他们给我提供了非常重要的帮助，指导和宝贵的建议。感谢指导老师帮助我选课题，在老师们的帮助下我的课题顺利开题。没有他们的努力，我也不能很快地整理自己地思路，解决编程出现地各种问题。

其次感谢站在我身后默默支持我的父母，四年的学费、学杂并不少。由于疫情的原因我也很少回家。

最后，值此论文完结之际，也是本人毕业之时。感谢辅导员三年和秋招时的帮助。感谢 8139 的三位舍友的四年关照。毕业在即，仅以此表达对老师们、同学们最衷心的感谢。

参考文献

- [1] 皮成.基于 Android 平台的即时通信中间件的研究与实现[D].西安电子科技大,2014.1-62.
- [2] 袁远.基于 Android 平台端到端即时通信系统的分析与设计[D].北京邮电大学,2012.1-67.
- [3] 吴亚峰.Android 应用案例开发大全第三版[M].北京.人民邮电出版社,2015.
- [4] 郭霖.第一行代码 Android 第三版[M].北京.人民邮电出版社,2020.
- [5] 严蔚敏,吴伟民.数据结构(C 语言版)[M].北京.清华大学出版社,2007
- [6] 余志龙,陈昱勋,郑名杰,陈小凤.Google AndroidSDK 开发范例大全 3[M].北京.人民邮电出版社,2011.
- [7] 纳德尔曼.Android 应用 UI 设计模式[M].袁国忠,译.北京:人民邮电出版社,2013.
- [8] 丰生强.Android 软件安全与逆向分析[M].北京:人民邮电出版社,2013.
- [9] 肖凯,张玉泉,陶智勇.基于 Reactor 模式的即时通信服务器的设计与实现[J].信息技术,2017(3):124-127,132.DOI:10.13274/j.cnki.hdzj.2017.03.031.
- [10] AndroidNetworkPacketMonitoring&AnalysisUsingWiresharkandDebookey[J]International Journal of Internet, Broadcasting and Communication, 2016
- [11] ArztS, RasthoferS, FritzC, et al. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps[J]. Acm Sigplan Notices, 2014, 49(6), 259-269.