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 Android Q 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.

**Key words**: 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用户需求分析 1

1.3服务器需求分析 2

1.4环境依赖 3

第二章 数据结构设计 4

2.1用户消息盒子 4

2.2自定义网络协议 7

2.3客户端其他数据结构 9

第三章 客户端设计 11

3.1UI设计 11

3.2通讯设计 11

3.3资源管理系统 11

3.4业务逻辑设计 11

第四章 服务端设计 13

4.1通讯设计 13

4.2数据库设计 13

4.3调度器设计 13

第五章 双端交互设计 14

5.1协议约定 14

5.2逻辑交互时序设计 14

第六章 设计评估 15

6.1功能评估 15

6.2兼容性评估 15

6.3性能评估 15

6.4设计评估 15

结束语 16

致 谢 17

参考文献 18

1. 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，且较电话更方便。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、微信等其早期会员功能可以支持服务器长期缓存聊天数据, 而后诞生的微信,则只采用短时缓存的方式,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)。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用户需求分析

1.2.1登录注册

用户输入用户名和密码, 对其进行检查和预处理, 首先需要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接收或拒绝都会有相应的提示。

删除用户一经某方提出, 数据库将直接删除一方对另一方的信息。 此处通过单方面好友实现了删除。

1.2.5日志

客户端由于使用安卓进行开发, Android 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。 其中:

Log.v 用于打印繁琐的意义最小的日志信息, 对应级别verbose

Log.d 用于打印一些调试信息, 这些信息是用于调试和分析问题的, 对应级别为debug

Log.i 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, 能够帮助分析用户行为的数据, 对应级别为info

Log.w 打印警告星系, 对应级别为warn

Log,e 打印错误信息, 一般用于进入catch语句后, 提示出现的错误, 是最高级别error

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。

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, 仅需声明即可。

客户端代码使用Kotlin编写, 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。 其次Kotlin内置了非常多的面向对象编程实现。 最后由于Java语言的特性, 最终Java虚拟机运行的是.class文件, 不在乎源码是什么语言, 所以Kotlin能完美兼容Java。

1.4.2服务端依赖

服务端采用Python3.10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。

服务器数据库采用Mysql8.0, 连接池采用了PyMysql, 简单易用。

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。 可用于通讯协议、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, 相比 xml、json、messagepack 在压缩比上有较大优势。 其序列化反序列化速度优于 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protocol buffers以及thrift较好，后两者都要预先定义schema并相对固定。

1. 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, 以便用户查看或知晓。 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(logn) 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后移。 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-1, 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。

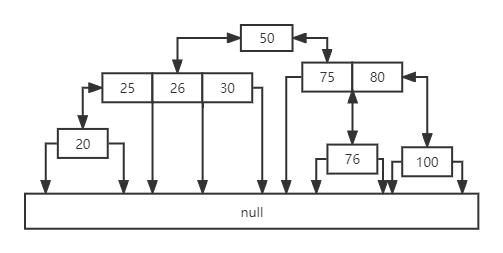


图2-1 2阶B\*树实现示意图(时间戳从大到小进行串联未画出)

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]](#footnote-1)]

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, 如果没有右数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:

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, 需要将左节点的右节点(包括空指针), 作为根节点的新左节点, 原先的左节点的右节点为原根节点。

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枝树(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()左右,

2.1.4LRU算法的实现

LRU算法是最主要的页面置换算法之一, 其根本思想在于, 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。

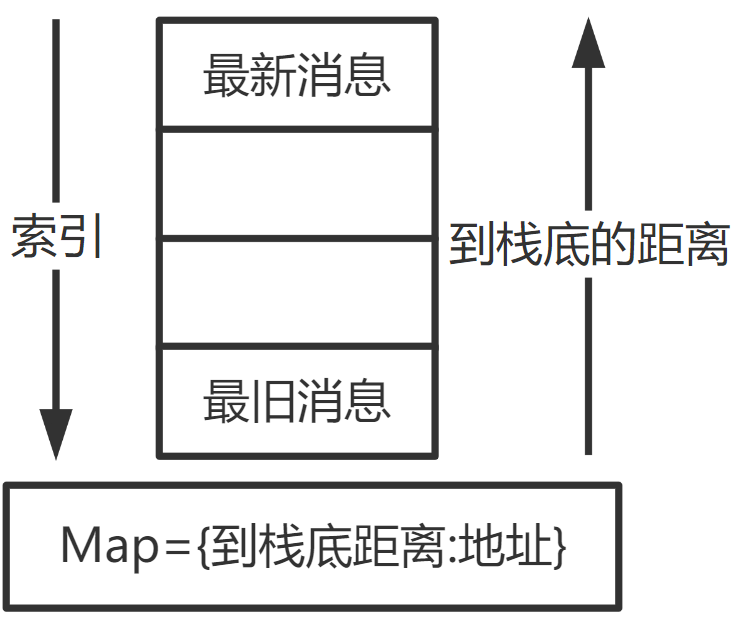


图2-2 LRU实现示意图

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, 也较小了内存开销, 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

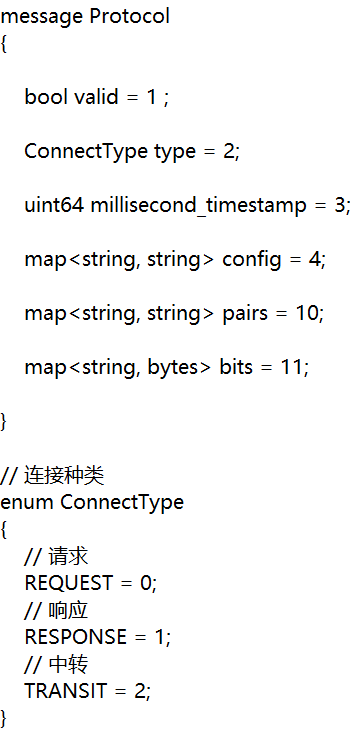


图2-3总体通讯协议

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, 将所有的数据、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。

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名字, 具体的文件在Bits中以名字映射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:

sealed class Result<out T : Any> {

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

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

}

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, 入伙成功, 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。

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。故设计中都以encoding代替加密。

1. 客户端设计

3.1UI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。

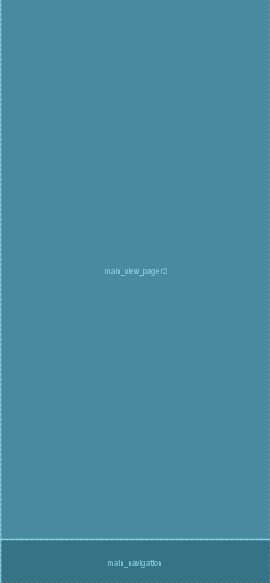
 

图3-1 用户主要界面(左)及布局设置(右)

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、按自定规则排序等。

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。

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3-2聊天界面(左)及设计(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

3.2.1Socket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。 用户第二种和服务器交换数据的方式是监听端口, 其主要的用途是当另一名用户发送了消息, 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函数中。 在C++中把关闭socket等释放资源的代码放在析构函数, Java在finnally, 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变成。 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。 经典的“建造者-指挥者”模式已经不太常用了，现在建造者模式主要用来通过链式调用生成不同的配置。 比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。

如代码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 协议建造者模式 (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。

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, 只能够通过数据库取资源的uri进行访问。 所以对数据库的统一管理显得非常重要。 本设计中, 除了发送图片不使用外部数据, 所以仅需

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, 服务器容易收到重放攻击, 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, 如果不是, 可以直接调用公用的 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时, 服务器将所有用户信息返回至用户。 如需加载其他内容, 如头像, 将采用懒加载的模式, 只有当用户到这个界面了, 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, 如果存在即为拉去成功, 不存在则代表着服务器也不存在该文件或连接出错。

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, 区别是前者由用户触发, 后者由服务器触发。 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。

1. 服务端设计

4.1通讯设计

4.1.1Socket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。

4.1.2Socket发送器

4.2数据库设计

4.2.1用户表

4.2.1用户关系表

4.3调度器设计

4.3.1

4.3.2用户关系表

1. 双端交互设计

5.1协议约定

5.1.1请求协议约定

5.1.2响应协议约定

5.1.3中转协议约定

5.2逻辑交互时序设计

5.2.1登录注册时序

5.2.2请求数据时序

5.2.3中转时序

1. 设计评估

6.1功能评估

6.1.1标题

6.2兼容性评估

6.2.1系统兼容性

6.3性能评估

6.3.1

6.4设计评估

6.4.1

结束语

结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例。

结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例结束语示例。

致 谢

我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢。

我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢我是真的谢。

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. 佘志龙,陈昱勋,郑名杰,陈小凤.Google Android SDK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. Android Network Packet Monitoring & Analysis Using Wireshark and Debookee [J] International Journal of Internet, Broadcasting and Communication,2016
11. Arzt S,Rasthofer S,Fritz C,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.

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 [↑](#footnote-ref-1)