# 整体需求

实现多人聊天的客户端服务端通信程序，要求具有群组功能和高可靠特性，支持灾备、负载均衡

# 主要功能

## 账户

1. 创建
2. 删除
3. 登录
4. 退出

## 好友

1. 添加
2. 删除
3. 获取好友列表

## 群组

1. 创建群组
2. 删除群组
3. 添加用户
4. 删除用户
5. 查看管理员
6. 修改管理员

## 消息

1. 发送文本消息（好友/群组）
2. 拉取离线消息
3. 发送文件
4. 历史消息存储、拉取、撤回

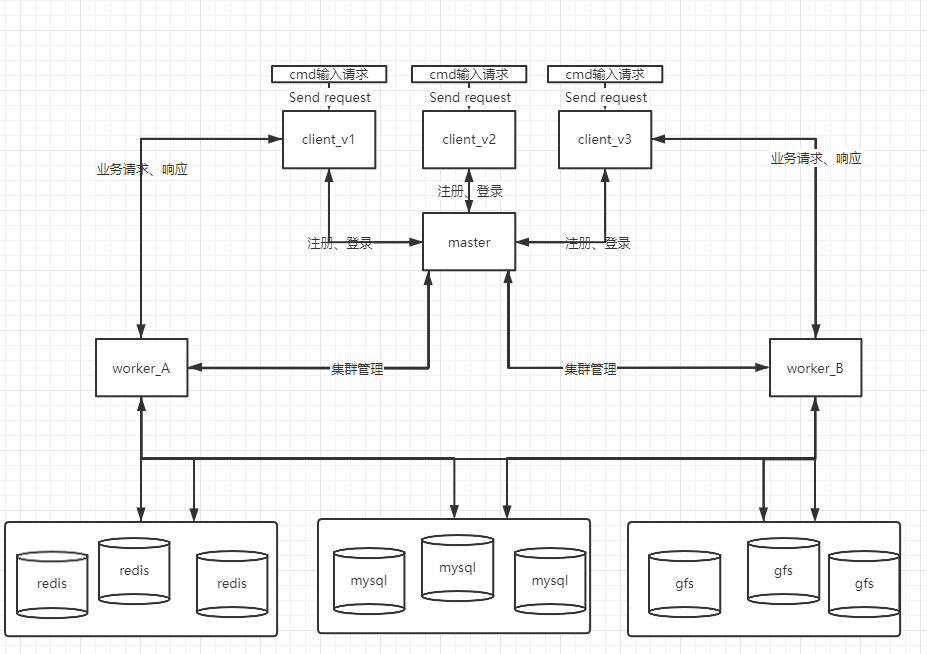
# 架构原理

后端实现：python3 flask/Django

数据存储：redis + mysql

前端实现: 暂无

## 整体架构



整个架构分为客户端、服务端（集群）、mysql集群、redis集群、gfs集群五部分：

**client：使用io复用模型，主要接收来自用户终端io输入、socket（tcp）数据包输入，并触发定时任务检查事务状态。**

客户端将用户输入的指令在客户端处理完后，封装到数据包中，数据包里至少包含事务id、版本号、指令类型和对应的实际数据（如登录的token和消息内容等），以json字符串形式组织，通过base64编码后发送给server集群。

**server：按一主多从的形式进行组织（一个master+多个worker）**，每次用户登录时，首先连接到master，master根据负载均衡算法给用户分配好worker，后续client端不再直接和master交互。

**worker也需要采用io复用，处理定时任务的同时，处理数据包**。在对数据包进行解析后，可以得到版本号和事件类型，分发给具体的事件处理函数进行处理

**mysql集群：主要用来作为消息存储库，存储历史消息**。

**redis集群：作为服务端公共缓存以及消息同步库**，负责存储账户信息、组信息、离线消息等等数据，实现worker间的信息同步。

**gfs集群：**用来存储**用户上传的文件和服务端系统日志文件**

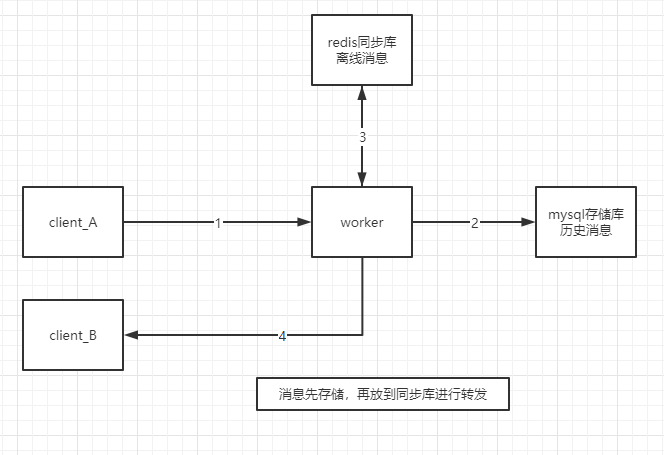
## 3.2服务端集群

如上节图示，**采用一个master+多个worker的架构**

master：添加、删除worker，检测worker状态，为用户分配、再分配worker

worker：处理实际业务

## 3.3消息存储



消息现在按待接收状态划分为两类：离线消息和历史消息，对应存储位置称为消息同步库（redis）和消息存储库（mysql），**统一采用先存储再同步的策略，避免旁路引发乱序**。

**离线消息：**用户离线时来自其他用户、群组的消息。

拉取离线消息属于高频操作，在登录、重连、收到消息序号不连续时都需要重新拉取离线消息，因此消息同步库使用redis集群实现，数据以接收用户为单位进行组织，群聊消息采用写扩散策略。需要对过期时间、离线消息最大阈值做出限制，只保留新的数据，其他的直接丢弃当做历史消息处理

**历史消息：**服务器上接收到的来自其他用户、群组的全部消息。

拉取历史消息需要用户端手动发出请求，相对频率较低但历史消息数据量很大，而历史消息的写入效率更加需要重点关注，考虑到后续可能需要支持各种检索匹配，选择使用数据库集群（mysql）存储。

这里给出一个存储思路：每个发送者的消息和群聊消息都用单独的表存放，从用户视角来看一次只需要一个会话的历史记录，群聊记录可以直接拉取，一对一聊天的历史记录只需要两张表的检索与合并，两张表需要自动记录插入时间（CURRENT\_TIMESTAMP）或者用全局的消息id做标志（不建议，可能严重影响性能），不然无法还原会话，如果后续要做水平分库还需要数据库服务器时间同步

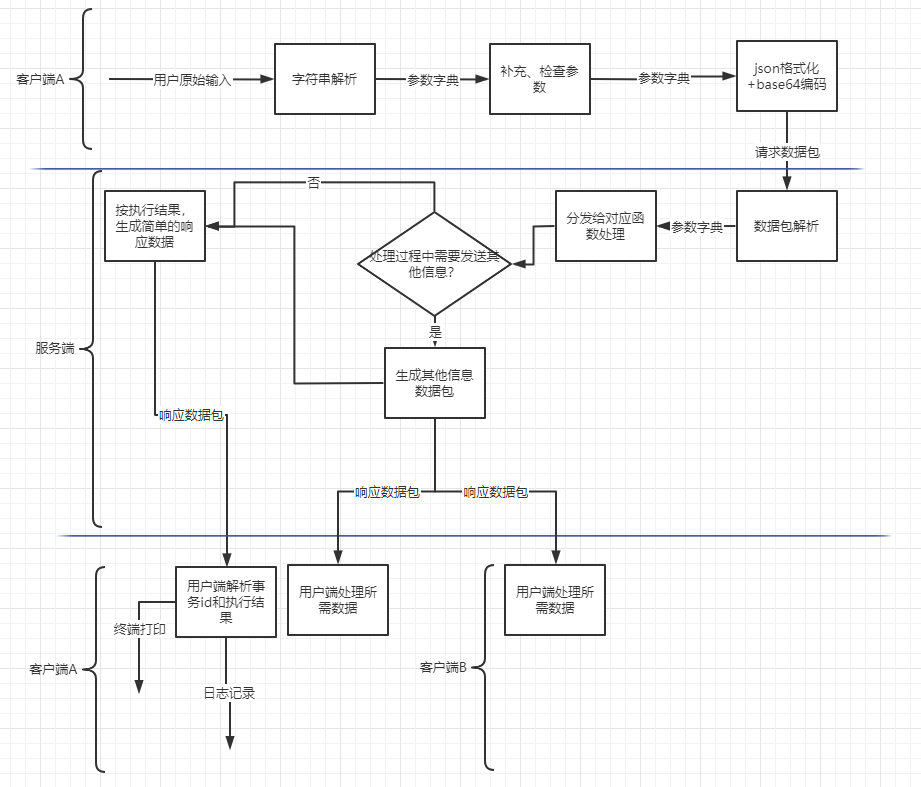
## 3.4请求处理

将**客户端发往服务端**的数据简单分为几类：

**业务管理请求、消息发送请求、文件传输请求、消息接收确认（用来告知服务端删除同步库离线信息，不单独讨论）**

### 3.4.1业务管理

主要指的是对账户、好友、群组的操作。

请求处理流程如下

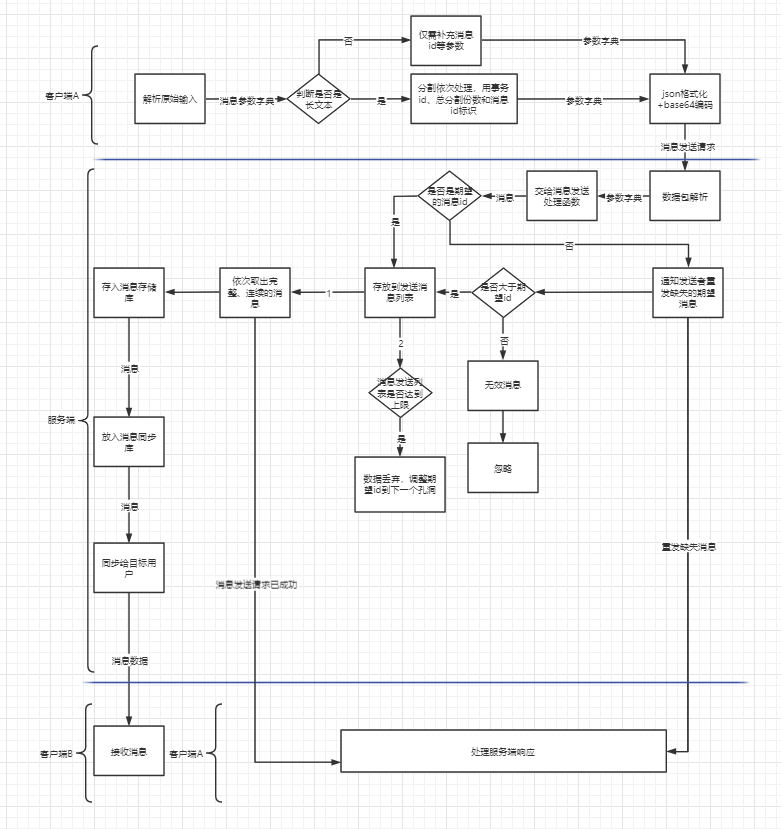
1. client封装管理请求成json字符串，按base64编码，通过socket（tcp）将数据包发送给负责该用户的worker（如果是登录，则会直接连默认的master）
2. worker识别到是业务管理类的请求，按照请求内容调用对应函数处理业务，如更新redis的用户信息、组信息、好友关系等，处理期间可能返回给用户所需的数据，如好友列表、worker信息
3. 最后worker将管理请求执行结果返回给client，通过gfs日志文件或者数据库对操作和结果进行记录和同步

### 3.4.2消息发送

虽然使用的是tcp协议，在正常网络下可以确保数据有序到达，但是数据依然可能丢失，另外需要确认长文本消息是否完整。考虑以下场景：长文本分3份发送，发送第二份时send成功代表发送方成功把数据复制到缓冲区，但接收端因为网络等因素没有接收到第二份，第三份正常接收到。此时应该要求发送方再次发送第二份数据，超时后再认为消息发送请求失败。

因此服务端会维护一个发送消息buffer。

处理逻辑如下：



补充图中未提到的几点：

1. **worker定时任务会检查所有发送消息列表**，检查同一事务id对应的最后到达时间和当前时间差，如果超过一定阈值则丢弃消息，处理发送列表后续消息
2. **消息同步库按接收者进行组织**
3. **目标用户如果在线，则同步离线记录给目标用户，用户确认收到后删除消息同步库的记录**

### 3.4.3文件传输

文件传输和长文本消息发送不同：

1. 因为大文件发送不能阻塞用户其他操作，所以客户端需要另起线程建立一个tcp连接用来发送文件。
2. 客户端这边维护一个待发送文件队列、发送开关变量、当前发送文件、当前发送文件块编号，首先客户端会向服务端发送一个文件传输请求，收到服务端的权限认证后客户端将文件追加到待发送文件队列，服务端放入允许接收列表
3. 文件传输线程会按照文件队列顺序往服务端发送文件，文件使用文件块编号，接收列表也不和消息混用
4. 服务端取出缓冲区数据后如果是期望的文件块编号，则直接追加写到gfs的对应文件中（按用户进行目录分隔），否则放到接收列表，并告知客户端停止发送其他文件数据，先重发缺失的文件块，接收到后再通知继续发送。
5. 接收完文件数据后，告知发送者发送成功，以消息的形式进行存储和同步，告知目标用户接收到了一个文件

流程图待补充。

# 存储数据结构

## 4.1redis

redis数据结构如下

用户名集合（set）：

achat:user

(username,...)

用户信息（hash）：

achat:{username}:user\_info

{"password":, "state":, "next\_recive\_id":,"token":，"worker":}

用户好友（set)：

achat:{username}:friends

(username,...)

离线消息上限（string):

achat:max\_offline\_message

"num"

用户待接收离线消息（set）：

achat:{username}:message\_set

(message\_id,...)

消息：

achat:{username}:{message\_id}

{"session\_type":, "sender\_name":, "content":}

群组集合（set）：

achat:group

(group\_name,...)

群组管理员集合（set）：

achat:{group\_name}:admin

(username,...)

群组成员集合（set）：

achat:{group\_name}:members

(username,...)

token=>user映射（string）：

achat:{token}

user=>token映射（string）：

achat:{username}

## 4.2 mysql

两个表分别存储点对点的聊天消息和群聊消息：

CREATE TABLE IF NOT EXISTS `user\_message\_table`(

`transaction\_id` CHAR(50) NOT NULL,

`frag\_num` INT NOT NULL,

`sender\_name` CHAR(20) NOT NULL,

`reciver\_name` CHAR(20) NOT NULL,

`content` TEXT(1024) NOT NULL,

`time` TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP,

PRIMARY KEY(`transaction\_id`, `frag\_num`)

);

CREATE TABLE IF NOT EXISTS `group\_message\_table`(

`transaction\_id` CHAR(50) NOT NULL,

`frag\_num` INT NOT NULL,

`sender\_name` CHAR(20) NOT NULL,

`group\_name` CHAR(20) NOT NULL,

`content` TEXT(1024) NOT NULL,

`time` TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP,

PRIMARY KEY(`transaction\_id`, `frag\_num`)

);

# 版本兼容

需要支持不同版本的客户端，不要求用户强制升级

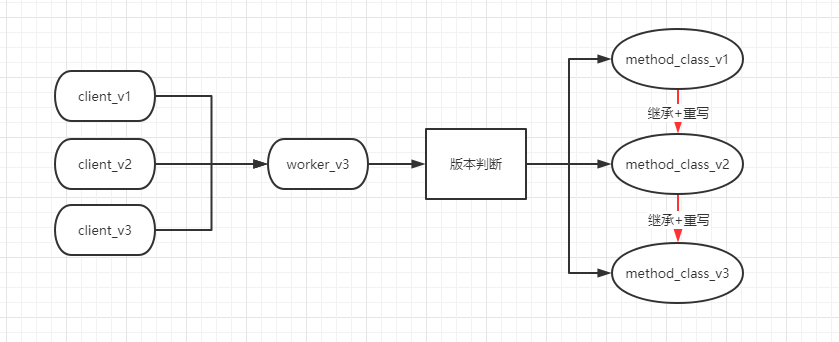
有以下方案：

方案一：

客户端**发送任何请求**时带个版本号；**服务端保留各个版本的方法**，用继承+重写的方式实现各个版本业务功能的类，外面加一层统一的服务端初始化、运行接口，由它先判断版本然后调用对应版本的分发处理函数，类似外观模式。

优势：后续扩展简单，某个api需要修改时只需要在新版本重写覆盖掉，而且新api不需要反复判断区分版本号

缺点：服务端代码量可能越来越多，到一定规模后需要放弃旧版本支持，向后合并



方案二：

客户端**登录**时带个版本号；服务端集群包含各个版本的worker，分配时按照客户端的版本号进行分配worker

优势：逻辑简单，不需要反复判断区分版本号

缺点：总代码量大，多个版本的服务端增加维护成本，还需要考虑各个版本用户的数目

方案三：

客户端**发送任何请求**时带个版本号；服务端的接口直接写成向后兼容的形式，区分版本处理

优势：代码总量小

缺点：代码逻辑臃肿，难以维护和扩展

**综合考虑选择方案一**

# 6.待改进方向

因时间关系，部分功能未完成

1.历史消息拉取，以及基于历史消息的会话还原

2.文件传输