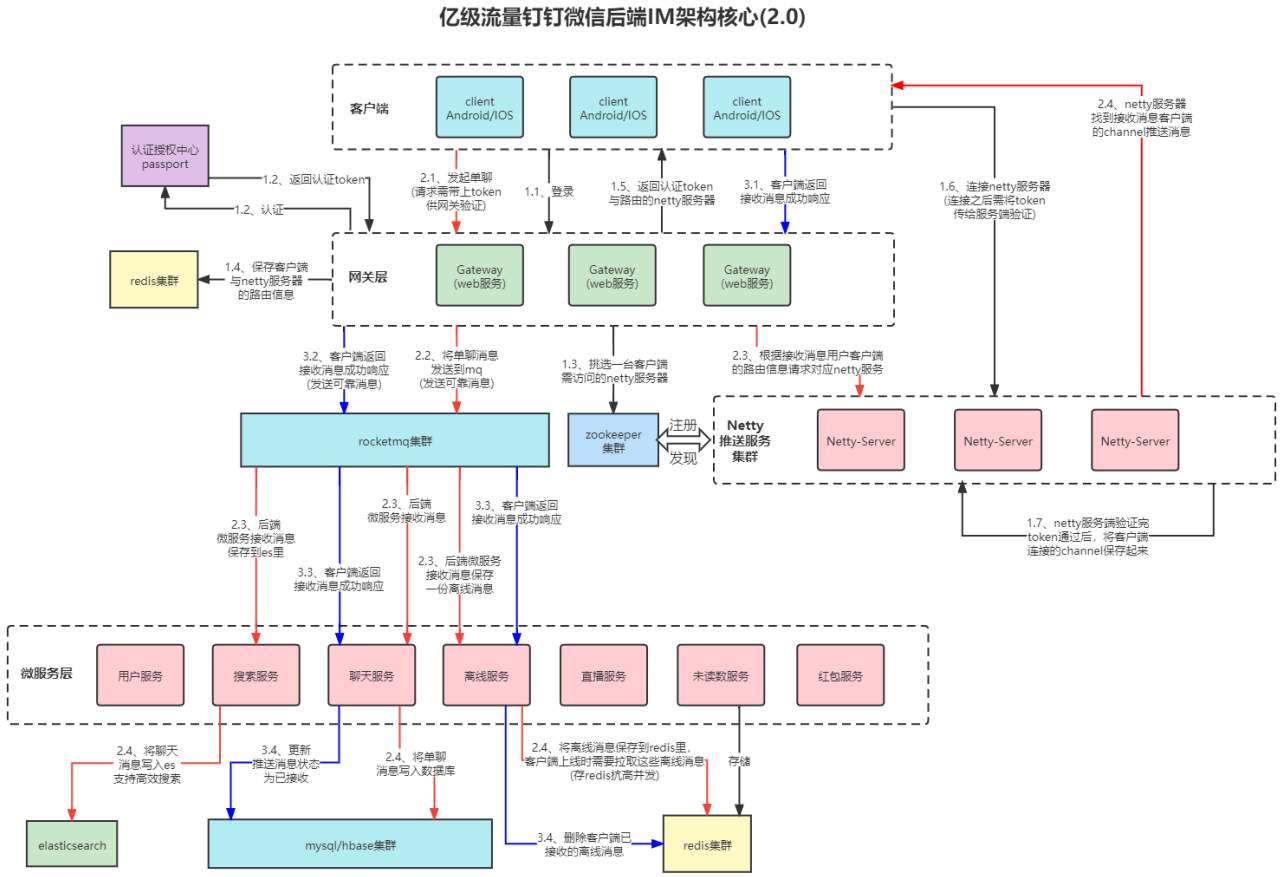
企业级IM系统核心架构图



如何保证聊天系统消息的可靠投递(不丢消息)

1. IM客户端发送消息如果超时或失败需要重发，客户端在发送消息时需要给每条消息生成一个id，IM服务端根据此id做好去重机制
2. 为保证服务端消息不丢失，我们可以使用Rocket MQ的可靠消息机制来保证
3. 通过客户端的ACK确认接收消息的机制来保证不丢消息

离线消息服务保证IM系统的高性能

1. 离线消息就是用户不在线时别人发给他的消息，到用户上线时这些消息需要接收到，因为用户上下线可能是非常频繁的操作，一般是在用户上线时会主动拉取服务端的离线消息，如果直接从数据库里拉，则会对数据库造成极大的压力，所以对于离线消息我们一般会选择一些高性能的缓存来存储，比如Redis，这样能抗住高并发的访问压力。
2. 当然Redis肯定是集群架构，而且会是很多节点，当然有同学也会担心这些离线消息肯定也是非常多的，Redis集群能存下吗，在大厂里Redis都是有很多节点的，可以存储很多T的数据，据说十年前新浪微博后端的Redis存储数据就已经达到几百T级别了，当然我们是可以设置一些存储策略的，比如，限制只存储最近一周或一个月的数据，然后再加一个存储消息的条数限制，比如一个用户的离线消息最多就存储最近的1000条。或者都按照存储条数的限制。
3. 因为本身用户上线后查看离线消息很少会把历史所有的离线消息全部看完的，我们就展示最近的一些离线消息，如果用户一直往上翻离线消息，后面的消息可以从数据库查询，这种小概率的操作让数据库抗下来是没问题的。

海量历史聊天消息数据存储方案详解

1. 消息存储结构参考数据库表结构设计
2. 发送消息处理
    - 用户1给用户2发送一条消息，需要在消息内容表里存储一条记录，同时需要在用户信箱消息索引表存储两条记录，一条是用户1的发件箱，一条是用户2的收件箱，为什么要存储两条记录，因为会存在消息的收发方各自删除记录的情况。
3. 查询聊天消息处理
    - 查看用户1跟用户2的聊天记录，首先可以先分页查询聊天消息索引的id，select mid,box_type from im_user_msg_box t where t.owner_uid = 1 and t.other_uid = 2 order by mid;(注意要分页查)，然后再for循环在im_msg_content表查每条消息内容展示。
    - 因为聊天消息数据巨大，我们肯定要考虑数据库的**分库分表**方案，im_user_msg_box表我们可以**按照 owner_uid 来分**，这样我们正常的聊天记录查询是不需要跨表查询的，im_msg_content表我们可以**按照 mid 来分**，因为查询消息基本都是按照消息id主键来查，性能非常高。
4. 关于表设计的解释
    - 这里解释下为什么将收发消息分为用户信箱消息索引表和消息内容表两张表来存储，因为很多时候消息内容会比较大，所以分成两张表来存储的好处在于，如果我们有时候只是需要读取一些消息收发的关系，而不关注消息内容的时候我们只需要查询用户信箱消息索引表即可，而不需要查询消息内容这种大数据表，对性能有一定提升。
    - 收发消息方可能存在各自删除消息的情况，所以要存储两份消息索引，因为我们将消息索引和消息内容是分开存储的，所以也不会导致消息内容这种大数据被存储多份。

合理选择Redis数据结构存储离线消息

1. 添加消息：zadd offline_msg_#{receiverId} #{mid} #{msg} // score就存储消息的id
2. 查询消息：zrevrange offline_msg_#{receiverId} 0 9 // 按消息id从大到小排序取最新的十条消息，上拉刷新继续查
3. 删除消息：zremrangebyscore offline_msg_#{receiverId} min_mid max_mid // 删除客户端已读取过的介于最小的消息id和最大的消息id之间的所有消息
4. 如果单个key消息存储过大，可以考虑按周或者按月针对同一个receiverId多搞几个key分段来存储

群聊数据收发机制读扩散与写扩散详解

1. 群聊我们目前的设计是基于**读扩散**，就是说用户在群里发一条消息只存一份数据，群里所有人都读同一份消息数据，这种方式比较简单，但是会有些问题，比如钉钉或企业微信的群聊消息已读用户列表功能就不太好实现了。

2. 写扩散机制：就是说用户在群里发一条消息会针对群里每个用户都存一条消息索引，然后再单独存储一份消息内容，这样可以针对用户是否已读做一些处理，但是写扩散有一个问题就是群的人数不能太多，否则性能会有问题，而且会有大量存储浪费，比如万人群聊，要是用写扩散，每个用户发一条消息，要存储上万条索引，这个对性能以及存储耗费太大。

基于Lua脚本保证消息未读数的一致性

1. 我们用的很多通讯软件都会有消息未读数的功能，就是软件右上角的红色数字，包括还有跟某人的消息会话的未读数，这个未读数功能是可以用redis来维护的，比如，张三给王五发一条消息，需要维护两

个redis key，一个是王五总的未读数key加1，一个是张三_王五的未读数key加1。这两个key的加1操作，我们是需要尽量保证它的原子性的，可以用lua脚本来实现。

2. 当然有同学可能会说总未读数可以不用单独维护，对所有消息会话的未读数求和就行了，注意，如果用户的消息会话比较多的话，那可能就会有性能瓶颈了，包括用户每个消息会话的未读数时刻在变化，在求和的过程中，可能之前读取的消息会话未读数又发生变化了，所以我们一般会单独维度总未读数。

3. 当客户端读了消息了，就给服务端发消息，服务端收到后更新未读数。未读数一般来说是在客户端自己去维护的，服务端的未读数更多是为了给**客户端多端数据同步**用的。

4. 对于群聊的未读数，一般可以针对群里每个人维护一个未读数key，比如用hash结构来存储，一个群里的所有用户的未读数可以用一个：hincrby msg:noreadcount:gid uid 1 （gid为群id，uid为用户id）

万人群聊系统设计难点剖析

1. 未读数更新高并发问题，对于万人群聊来说，1个用户发1条消息，可能会触发群里所有用户的未读数更新，相当于更新两万次redis里的未读数，如果这个群比较活跃，每秒假设有100人发消息，每人发1条，那意味着光这一个群每秒钟会对redis操作两百万次，这会对redis造成巨大的压力，因为未读数其实主要是在客户端自己维护的，服务端维护主要是为了多端同步，所以我们可以不实时更新服务端的未读数，而改为定时批量更新，比如每5秒甚至更长时间更新一次redis里的未读数，这5秒内的未读数更新可以在未读数服务的内存里更新，到了5秒了，再统一往redis里刷一次，在其他端要同步的时候可以触发一次内存未读数刷新redis的操作之后再同步。当然中间可能会出现未读数服务宕机导致丢失部分未读数，一般来说业务是能够允许的。

2. 万人群聊还有一个高并发难题，按我们现有架构，1个用户发1条消息，要给群里1万个用户转发消息，意味着每秒要查询1万次redis里的netty服务路由信息，如果这个群比较活跃，每秒有100人发消息，每人每秒发1条，意味着1秒钟要查询100万次redis，这还只是1个群了，如果有成千上万的群，那对redis也会造成巨大的压力，这个问题与后面要讲的百万人在线直播间发消息很类似，都有高并发问题，我们放在后面一起来讲。

百万在线直播互动场景设计难点剖析

1. 上面提到的万人群聊第二个问题，放在这种百万人在线的直播间，问题会更严重，试想下，假设这个直播间每秒有100人发消息，每人每秒发1条，意味着每秒要查询1亿次redis，redis集群就算机器再多也是扛不住这种压力的，那我们之前的方案在这种万人群聊以及百万在线直播间的场景下肯定需要做优化了。

2. 这种情况我们其实可以不经过redis，直接把消息投递给所有的netty服务器，假设有20台netty服务器，对于百万人的直播间，每台netty服务器应该有5万左右的直播间用户连接，我们可以让每台netty服务直接将消息推送给自己服务器上对应连接的直播间用户。当然这种用户连接是比较理想的情况，有的时候可能会出现某些netty服务器几乎没有直播间的用户连接，那么这些netty服务器收到消息后直接丢弃就行，不会浪费什么系统资源。当然这种情况其实可以让所有的netty服务器监听mq的消息。

3. 在netty服务器上可以维护好直播间或万人群聊里连接到本台服务器的用户列表缓存，这样只要收到直播间或群聊消息，直接根据直播间号或者群聊id找到对应用户连接channel推送消息。

熔断限流机制保证消息收发核心链路高可用

对于大型的直播间或万人群聊，因为一些热点事件，或者大V出镜直播，会导致消息量暴增，以至于超过服务器能处理的压力范围，这种时候我们是需要做一些限流措施的，可以丢掉部分消息，以确保整个系统的稳定，这

种处理一般来说业务也是允许的，而且如果每秒大量的消息推到客户端，客户端不一定能及时处理下来，一般客户端在直播间，每秒接收几十条消息已经快到极限了。