# 消息系统示例

消息系统是 IM 的核心，也是相对比较复杂的部分。

这个示例渐进的实现了一个采用推拉结合，读写分发混合的消息系统，借助 WebSocket 实现推送。

> 示例代码后端使用 Node.js （Express），前端使用 Vue 2。

## Phase 1 - 最简单的 IM

在这一阶段，我们先解决消息收发的问题。

### Level 1 - 基本的消息收发

作为入门，我们先实现一个最简单消息收发的模型。

#### 模型抽象

介绍一下这个阶段我们需要使用到的概念：

- 用户、消息：字面意思。。。
- 消息链条：由消息组成的一个有序列表

当我们思考如何写代码时，我们通常会先去现实世界里找一个例子，然后分析它的工作方式，IM也不例外。

想象一个传纸条的场景，有若干个同学想通过传纸条的方式交流，一个方案就是找一张很长的纸，然后一行一行的往下写。

但是这样有一个问题，这张纸在一个时刻只能在一个人手里，其他人如果想看这张纸，就需要先把纸拿过来，所以我们稍微改进一下这个方案：

- 指派一名同学专门负责管理这张纸；
- 如果其他同学如果希望在这张纸上写内容，那么就自己准备一张纸，写上内容，交给负责管理的这个同学。这个同学收到纸条后，把交上来纸条的内容抄到这张大纸上；

- 如果其他同学希望知道有没有新内容，就去不断地问这个管理的同学，如果有新内容，管理的同学就把新内容抄在另外一张纸条上，发给来咨询的同学，最后其他同学本地排列一下所有发下来的纸条，就是所有的记录了。

### 代码实现

在理解了模型的工作方式之后，就可以动手开始写代码了。

整体并不复杂，基本上就是在 Server 维护了一个消息链条（普通的 Array），然后针对这个链条，实现两个操作：

- Send，向这个链条尾部追加一条消息
- Pull，从链条的指定位置开始，获取若干数量的消息

## Level 2 - 多会话聊天

前面虽然我们已经实现了基本的消息收发，但是目前所有人都是共用一个消息列表的，相当于一个大型的聊天室，那么如果我们想实现比如说，拉个小群聊天，或者一对一的私聊，上面的模型就需要改进了。

### 模型抽象

在这里我们引入几个新的概念：

- 会话，指若干个用户的集合，可以有一个或多个用户（私聊可以理解为只有两个用户的会话，群聊则是多个用户）
- 用户消息链：指关联到一个用户的消息链条，下文简称用户链
-

还是一些同学们在传纸条，这个时候需求变成了希望能有私聊纸条和群聊纸条，还是一个管理的同学和一些需要交流的同学，我们只要把模型改成这样就可以解决问题：

- 如果有一些同学希望加入某个群，那么就去找管理的同学登记。管理的同学需要维护一个表格，上面记录了有哪些群，每个群里有哪些同学；
- 由于一张纸不够用了，现在需要给每个需要交流的同学都准备一张独立的纸。在有一个同学交一张纸条上来的时候，询问这个同学需要发到哪个群里，然后找出这个群里所有的同学，把这个纸条上的内容抄到每一个同学单独的纸上；
- 每个同学还是不断地问管理的同学有没有新内容，但和之前的不同，现在管理的同学只会去看这个同学自己的纸。

下面用一个实例来分析这一阶段我们需要做的修改。

例如，现在有一个会话1，其中有用户A、B、C；会话2是B和C的私聊会话，会话3是A和D的私聊会话。

现在系统中总共产生了15条消息，其中消息1-5是会话1的群聊，6-8是会话2的私聊，9-10是会话1的群聊，11-15是会话3的私聊，下面列出一个表格，如果这个用户应该接收到这条消息，那么就打一个√：

| 消息 | 会话1 | 会话2 | 会话3 | | 用户A | 用户B | 用户C | 用户D |
|---|---|---|---|---|---|---|---|---|
| 1 | √ | | | | √ | √ | √ | |
| 2 | √ | | | | √ | √ | √ | |
| 3 | √ | | | | √ | √ | √ | |
| 4 | √ | | | | √ | √ | √ | |
| 5 | √ | | | | √ | √ | √ | |
| 6 | | √ | | | | √ | √ | |
| 7 | | √ | | | | √ | √ | |
| 8 | | √ | | | | √ | √ | |
| 9 | √ | | | | √ | √ | √ | |
| 10 | √ | | | | √ | √ | √ | |
| 11 | | | √ | | √ | | | √ |
| 12 | | | √ | | √ | | | √ |
| 13 | | | √ | | √ | | | √ |
| 14 | | | √ | | √ | | | √ |
| 15 | | | √ | | √ | | | √ |

观察上表，很容易发现一个规律，我们只需要关注消息在发送时，这个用户有没有在会话里即可。在收到上行的消息时，我们可以把这个消息复制 N 份，然后向会话里的所有 N 个用户都分发一份这个消息的副本，每个用户只需要去拉取自己获得的消息副本，就可以获得这些消息了。

用户收到的这些消息副本，可以构成一个消息链条，我们称之为用户消息链，简称用户链。

### 代码实现

我们还是需要 Send 和 Pull 两个接口，接口的行为需要做一点点的调整：

- Send：将消息追加到消息目标会话中每个用户的用户链上；
- Pull：拉取特定用户链上的消息。

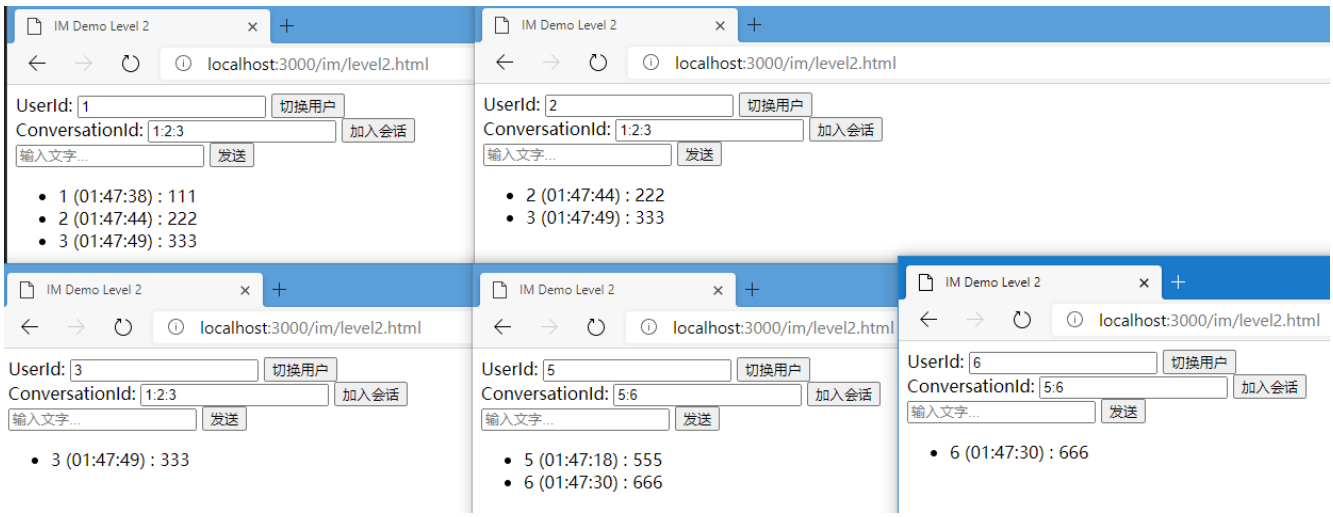
然后还需要一个新的接口 Join，即把某个用户添加到会话中。

### 实验过程

我们打开5个浏览器窗口，访问我们的网页。

让用户1、2、3加入会话 `1:2:3`，作为一个群聊；用户5、6加入会话 `5:6` 作为一个私聊，然后各个用户随意的发几条消息。

可以看到我们的代码工作符合预期，1:2:3 会话的用户和 5:6 会话的用户可以各自独立的聊天。




## Level 3 - 会话历史消息

我们都用过微信，微信有个设计，就是后面加群的人看不到历史消息，我们在 Level 2 做出来的 IM 似乎也刚好复现了这个设计，那么如果我们想看历史消息，该怎么继续改进？

### 模型抽象

我们再引入一个概念：

- 会话消息链：指关联到一个会话的消息链条，下文简称会话链

然后还是画一个类似 Level 2 的表格，来模拟一个加群的过程，假设A和B已经在群里聊了5条消息，此时C加群，并且再聊了3条消息，然后B退群，A和C发了3条消息，B之后又重新加群，并且又发了3条消息，A再退群，最后发了2条消息。

这个表格大概就是这样的：

| 消息 | 会话1 | | 用户A | 用户B | 用户C |
|---|---|---|---|---|---|
| 1 | √ | | √ | √ | |
| 2 | √ | | √ | √ | |
| 3 | √ | | √ | √ | |
| 4 | √ | | √ | √ | |
| 5 | √ | | √ | √ | |
| 6 | √ | | √ | √ | √ |
| 7 | √ | | √ | √ | √ |
| 8 | √ | | √ | | √ |
| 9 | √ | | √ | | √ |
| 10 | √ | | √ | | √ |
| 11 | √ | | √ | √ | √ |
| 12 | √ | | √ | √ | √ |
| 13 | √ | | √ | √ | √ |
| 14 | √ | | | √ | √ |
| 15 | √ | | | √ | √ |

我们的期望是：

- 在第6条消息C加群时，能看到第1-5条消息
- 在第11条消息B加群时，能再看到第8-10条消息

观察一下可以发现，如果我们把消息再按会话维度保存一次，然后在发生加群、退群等操作时，从会话上把这些消息给补回来，问题就解决了。

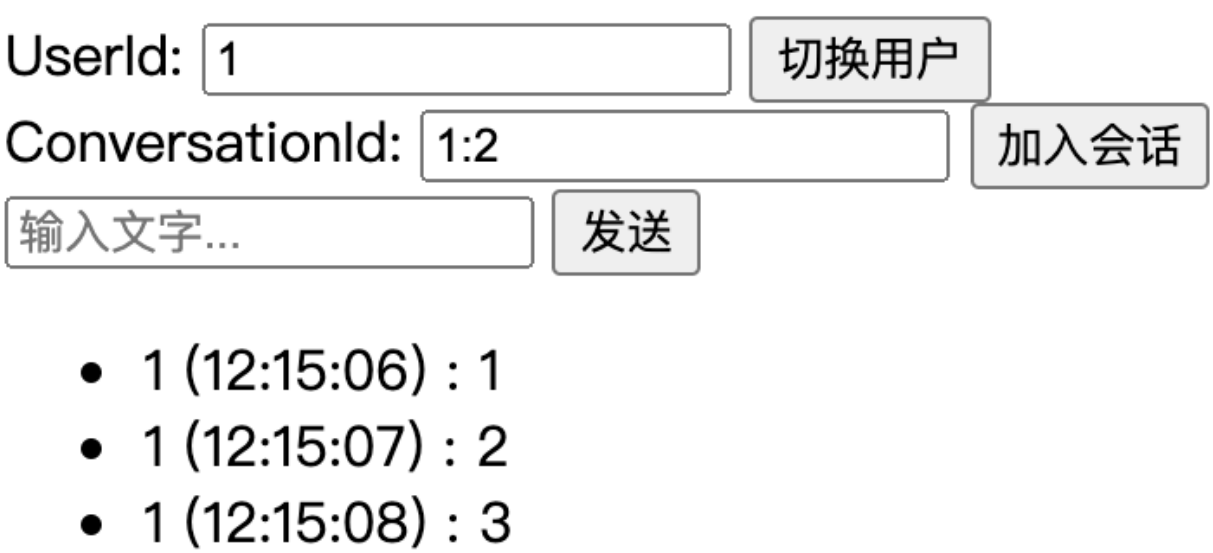
## 代码实现

我们需要一个新的接口：PullConversation，即按会话维度拉取历史消息，在前端我们需要把这些拉取下来的消息，和自己用户链上的消息做一个合并，去掉自己用户链上已经存在的那部分消息，我们可以先简单的使用时间戳去重，更好的方案是针对每条消息都生成一个独立的 ID，然后使用这个 ID 去重。

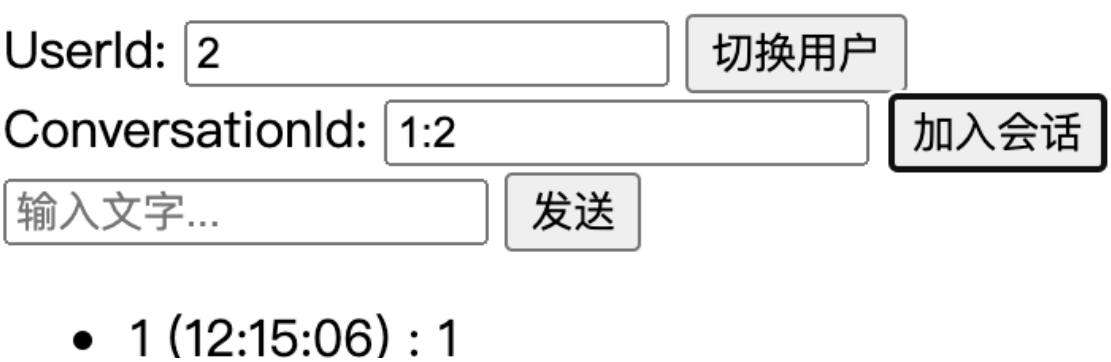
由于已经进行了加群操作，因此 PullConversation 只需要在进群的时候调用即可，后面的消息都可以通过拉取用户链获得。

## 实验过程

我们先以用户1的身份，向会话1:2里发几条消息：




然后切到用户2，加入会话1:2，可以看到历史消息都顺利加载出来了。



- 1 (12:15:07) : 2
- 1 (12:15:08) : 3

## Phase1 总结

在前面的几个阶段中，我们从逐步构建起了一个能收发消息的 IM 系统。这个 IM 是一个最简单的模型，但却是微信、QQ、抖音私信、飞书等超大型 IM 背后的基础。

如果了解读分发和写分发的概念，很容易可以知道，在 Level 1，我们构建了一个读分发系统，每条消息只写入一遍，而在 Level 2，我们构建了一个基于写分发的系统，即每条消息都会被重复写多次到用户链上，在 Level 3，我们尝试综合了前面的两个模型，即每个消息被单次写到会话链上，同时多次写到用户链上。通过读写分发的结合，我们很好的满足了在线消息+历史消息的需求。

由于是一个简单的模型，我们忽略了很多其他和消息流动（也叫消息分发）无关的信息，例如：

- 用户系统，即登录、注册、用户个人信息保存等，在我们的示例中，用户被抽象为一个唯一的字符串
- 会话信息，例如群名，群头像，建群时间等
- 群成员操作，例如设置管理员，拉人进群，踢出群等
- 消息持久化，我们现在所有的数据都是保存在内存中的，服务端程序重启就会丢失

# Phase 2 - 高阶 IM 操作

主要介绍思路，具体可以看代码实现。

## Level 4 - 消息索引

在前面我们一直没有详细介绍一个很关键的变量cursor，在这个部分我们会对cursor做一点点改造，来满足我们后面的一些需求。

由于这个简单的 IM 不存在高并发的问题，我们可以先使用每条消息的创建时间作为消息的唯一索引字段。注意到我们在拉取用户链时传入了cursor参数，这个参数在之前是用户链的长度，我们把它修改成时间，然后调整原来 slice 的方式为 filter。

这里还有一个优化点，例如说一个群已经存在了一个月，里面有非常多的聊天记录，按照我们Level3的处理，我们需要把这个群的所有消息都拉取回来，通过利用消息索引，我们可以控制只拉取某个阶段的消息。

## Level 5 - 已读状态

在前面我们使用时间戳构建了一个消息索引，还需要考虑一个场景，即已读未读状态，对应的就是未读数（某个会话有x条未读消息）。

未读消息可以在前端计算完成，思路就是保存一个 readIndex，值为消息的索引，指向当前已经读到的消息，然后只需要计算一下 不是自己发送的消息 并且消息的索引大于 readIndex 就可以得到未读消息数了。

我们需要一个新的接口 markRead，它的作用是上报当前用户在当前会话的 readIndex；在 PullConversation 接口中，我们需要下发这个 readIndex 值到前端。

## Level 6 - 命令消息

命令消息的意义在于，同时登录多个客户端时需要同步状态，例如 Level 5 中的已读状态，在电脑和手机上同时打开QQ，然后在手机上阅读了一些新会话，此时电脑上的QQ这些未读会话也会变为已读。

以已读状态为例，我们可以用这种方式实现：

- 在服务端接收到 markRead 操作时，更新服务端保存的 readIndex 值，并创建一条新消息，这条消息对用户不可见，携带服务端的 readIndex 值，并写入到这个用户的用户链上分发给用户；
- 用户在接收到这条消息时，读取消息中的 readIndex，更新到本地，再刷新这个会话的未读数。

用同样的思路，我们可以实现消息撤回、删除等，对于撤回就是构建一条特殊的命令消息，含义为撤回索引为 xxx 的消息，并把这条消息分发给会话中所有的成员，前端接收到这条消息时，就给本地自己的这条消息打一个撤回的标记；删除则是只分发给自己的一条含义为删除索引为xxx消息的命令消息。

### Phase 2 总结

在这一阶段，我们为IM系统添加了一些新的重要概念，如消息的索引位置和状态等。在有了这些之后，我们就完成了一个 IM 系统所有的核心能力，它已经可以很好的支撑各种常见的IM需求了。

但我们一直忽略了一个重要问题，我们一直在采用拉模式（轮询）来处理新消息，并且每次都需要处理全量的历史消息，对于客户端和服务端的压力都非常大，因此在第三阶段，我们将引入一些优化的措施，例如借助 WebSocket 实现消息的实时推送。

## Phase 3 - IM 优化

### Level 7 - 实时消息推送

基于轮询的机制可以满足 IM 收发消息的要求，但实时性被限制到了轮询的间隔长短上。为了解决这个问题，我们需要引入更加实时的推送方案。

我们构建一个长连接系统，它有这么几个接口：

- registerClient，承接前端的 WebSocket 建连请求，保存一个从 userId 到 WebSocket 连接的映射关系；
- pushMessage，将特定的结构推送到某个 userId 的前端页面。

然后我们在消息写入用户链时，调用一次 pushMessage 直接把这条消息推到前端即可，前端也不再需要轮询 pull 接口了。

然后我们需要考虑一个情况，假如这条消息没推下去，怎么办？这时我们需要依赖拉接口来把这条消息给补回来。

在推送的时候，我们不仅要推送消息本身，还需要推送这条消息对应前一条消息的cursor，在收到这条消息时，我们对比推送携带的cursor和本地的cursor，如果两个cursor不一致，那么说明中间发生了消息丢失，调用一次pull接口即可把这些消息补偿回来。

## Phase 3 总结

在这一阶段我们通过 WebSocket 实现了消息的实时推送，并通过我们之前实现的用户链机制来保障消息无法实时推送时仍然不会丢失。

IM 的消息系统到这里就基本打通了，消息的生命周期可以简单的归为下面这样一个流程：

- 消息发送，客户端上行请求；
- 消息分发，服务端根据一些规则来将消息写入用户链和会话链；
- 消息送达，服务端将消息通过推送，或客户端通过拉取等方式将新消息保存到本地；
- 消息处理，根据消息的形态内容等，在客户端执行一系列的操作。