IM分享

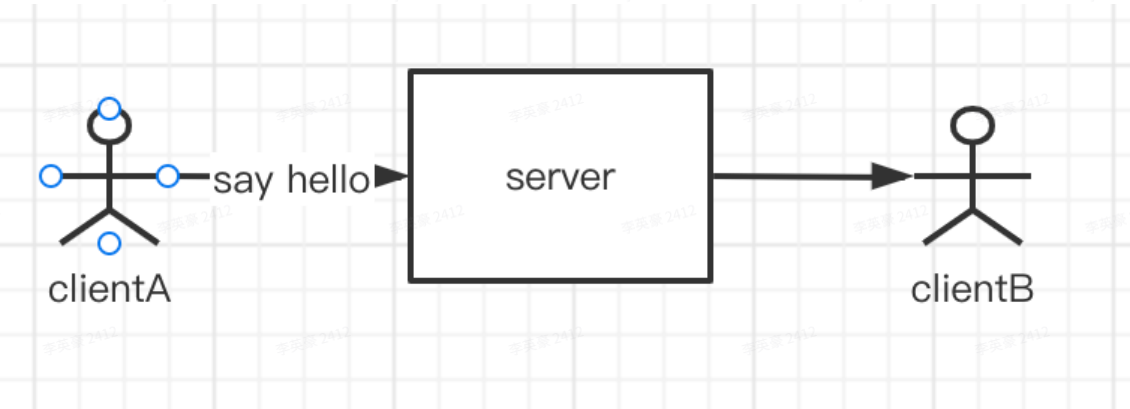
> 📌 IM即即时和通讯，顾名思义核心就两个即时性和通讯可靠性，这也就奠定了IM能力的核心诉求，就是要保证消息的即时触达和可靠通讯。
>
> 1. 怎么保证消息可靠传输?
> 2. 怎么保证消息及时触达?

可靠通讯

上面我们说了IM的核心诉求之一就是消息的可靠通讯，那么怎么保证消息能够可靠的发送给目标就是我们需要去考虑的一件事情。我们先从简单发送消息开逐步去认识如何实现可靠的通讯

普通的消息发送

我们的普通消息发送比较简单，一般就是客户端发送一个 `msg` ，服务端收到这个 `msg` 后推送给其他端，如果不在线则推送 `push` 去触达，如下：



这样就完成了一条消息从 `client-A` 简单发送到 `client-B` ，如果网络正常的情况下，上面的流程是没有问题的，但是网络必然是会存在异常波动的那么这个简单地流程就无法保证其中消息的可靠性，因此我们需要 `ACK` 帮助我们完成这件事情

ACK的出现

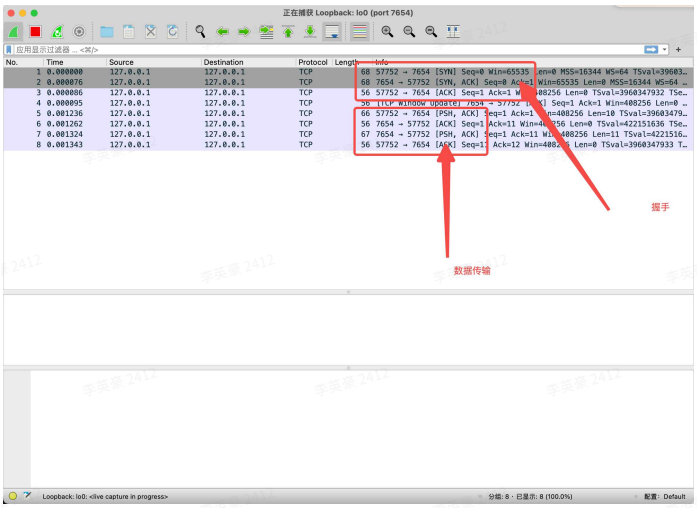
因为网络环境是不可控的，会出现各种各样的问题，那我们考虑以下两个场景出现的问题：

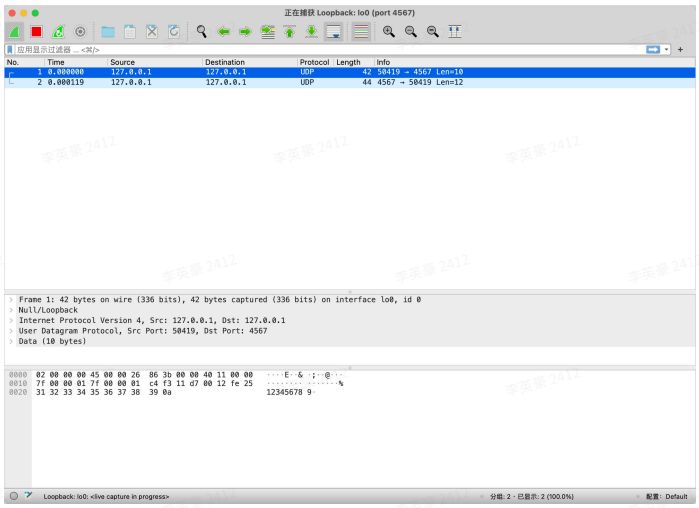
1. 网络崩溃 `server` 没有收到消息
2. 网络崩溃 `clientB` 没有收到消息

我们的客户端是基于 `websocket` ，服务端一般基于 `socket.io` 它们类似于 `UDP` ，因为发消息无状态所以无法保证消息的成功发送、推送，它们只管发不管对方有没有接收到（因为没有 `ACK` ），但

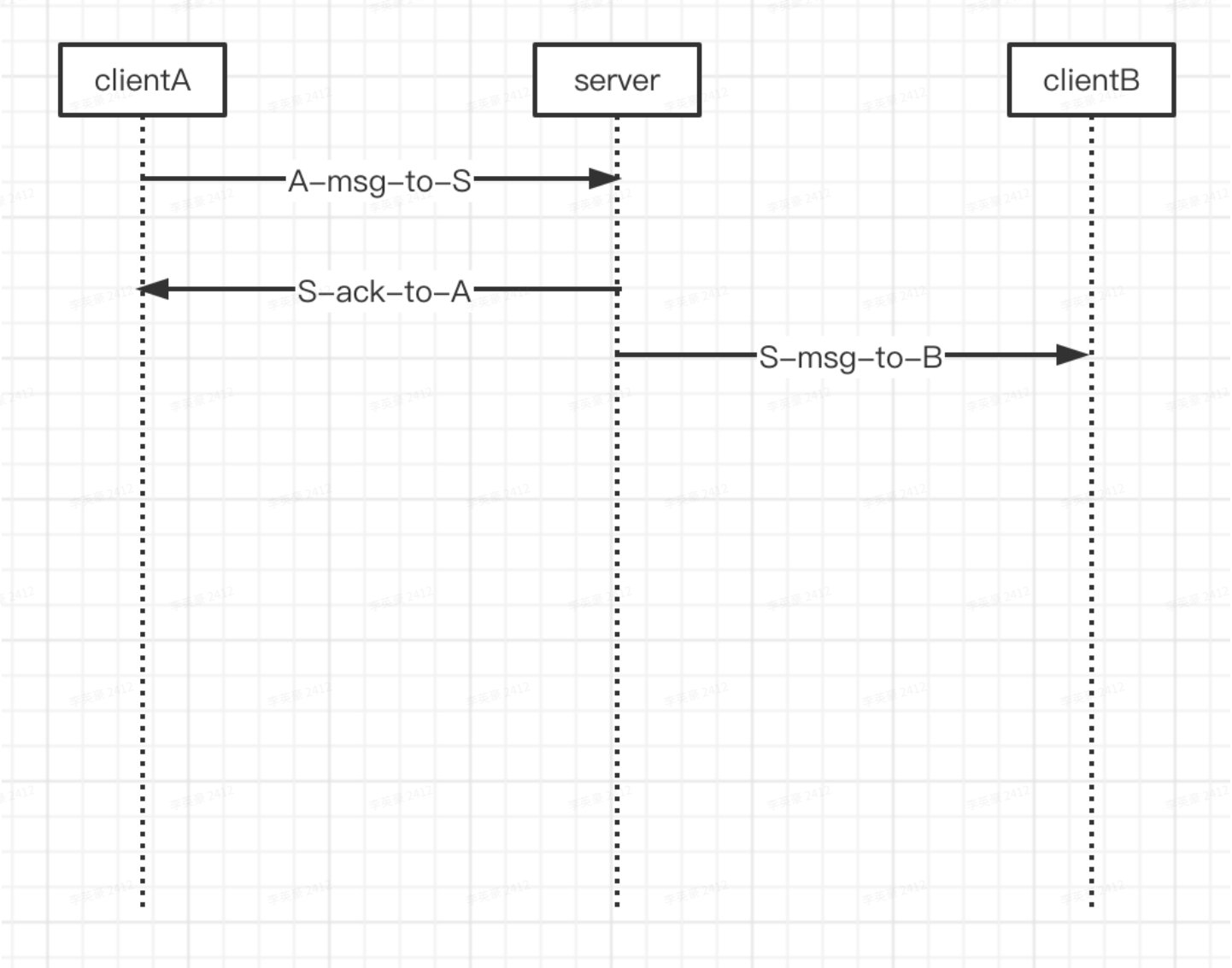
是 `TCP` 是有状态的可以知道请求的成功或失败，那我们可以仿照 `TCP` 实现一套简易的保证消息成功发送的机制

> TCP vs UDP





1. clientA发送消息 `A-msg-to-S`
2. 服务端收到这条msg，并给客户端返回 `S-ack-to-A` ，并给 `client-B` 推送消息 `S-msg-to-`
3. `clientA` 收到 `S-ack-to-A` 确保消息成功发送



这样的话在客户端维护一个已发送的消息队列，假定在一定的时间内没有收到该消息的 `ACK` ，我们就可以认定该消息发送失败尝试重发、给用户提示，这种方案很大程度上解决了一部分问题，但是这个方案还存在一定的缺陷：

1. 即使 `clientA` 收到了 `S-ack-to-A` 也不能确保 `clientB` 一定收到了这条消息
2. 假如 `server` 在给 `clientB` 推送的时候因为网络原因断掉了， `clientB` 还是无法可靠的收到消息

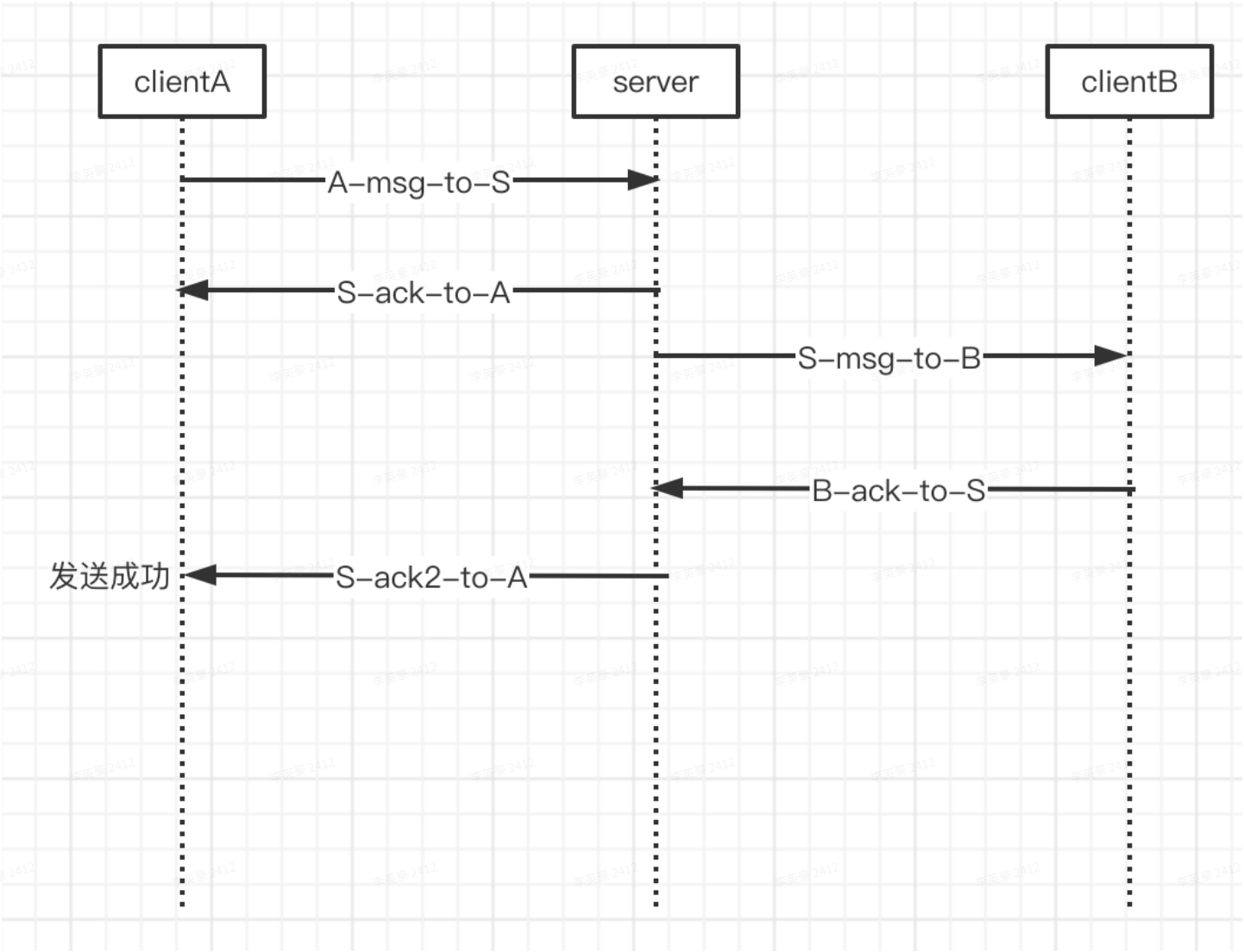
六种报文的出现

经过上面的尝试我们发现，即使clientA收到了服务端的 `ACK` 也不能保证消息的可靠传输，那么想要保证消息的可靠传输我们需要怎么做呢?

我们需要有六种报文状态：

1. `clientA` 发送消息到 `server` `A-msg-to-S`
2. 服务端回复 `ACK` 到 `clientA` `S-ack-to-A` 来确保 `clientA` 消息发送成功，并给 `clientB` 推送消息 `S-msg-to-B`

3. `clientB` 回复 `ACK` 到服务端确保服务端知道 `clientB` 已收到消息 `B-ack-to-S`
4. 服务端回复 `clientA` `S-ack2-to-A`，确保 `clientA` 知道消息真正的发送成功了



上面的流程可以准确地保证消息的可靠性，但是但凡中间任何一个环节出现问题都很难确定消息是否可靠的被传输，因此我们需要做很多错误处理

消息的重发&超时处理

我们在最上面说过我们可以通过判断 `clientA` 是否收到服务端的 `ACK` 确保消息是否发送成功了，但是其中还存在很多问题：

1. `S-msg-to-B` 可能会失败
2. `B-ack-to-S` 可能会失败
3. `S-ack2-to-A` 可能会失败

综上所述我们的消息可能会在各个环节因为不同的原因导致 `clientA` 无法收到正确的ACK，那我们单单只是靠上面的办法做时会有问题的，主要体现在：

1. `clientB` 收到了消息，但是服务端不知道
2. `clientB` 收到了消息，但是 `clientA` 不知道

这种情况下 `clientA` 是需要做重试或者提示用户，会导致 `clientB` 收到重复的消息，那么这种情况下我们要怎么处理呢?

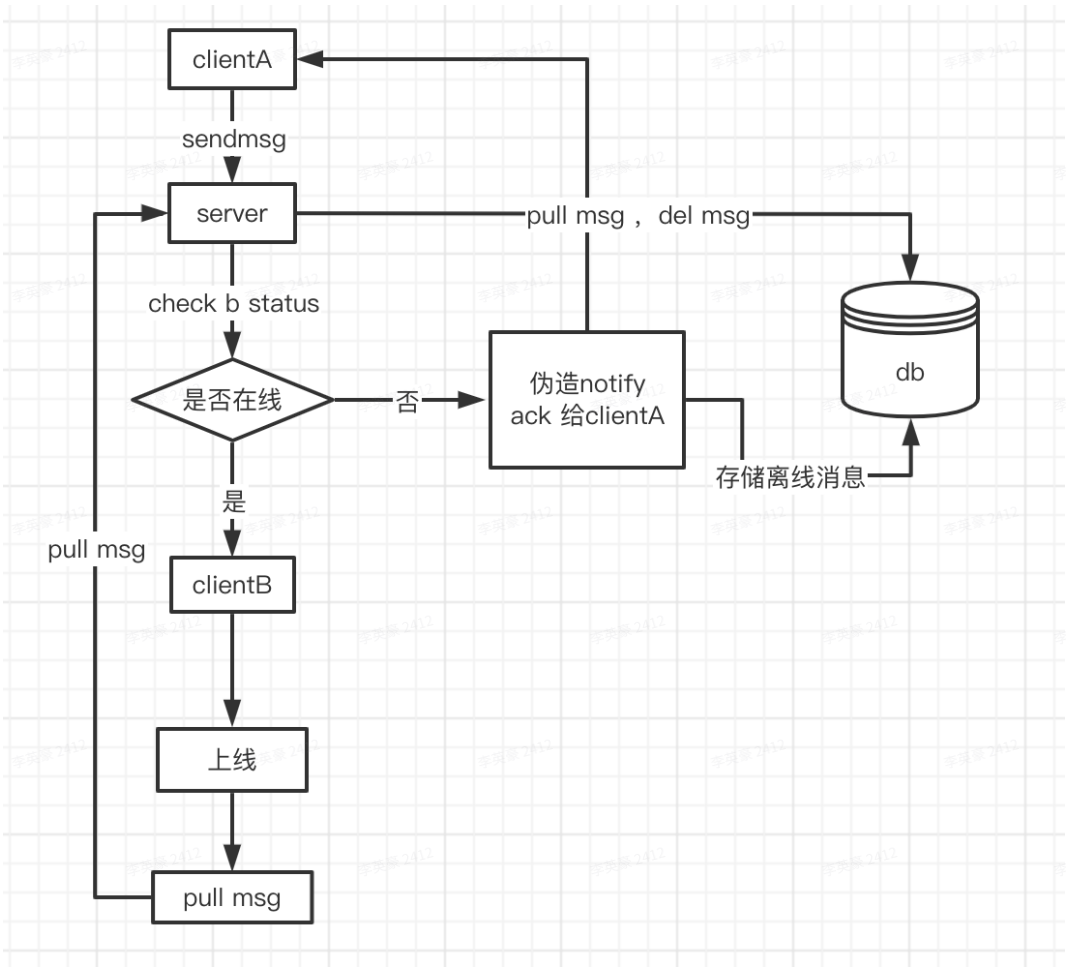
去重：这里的处理方法也很简单，我们每次发送给 `clientB` 的消息保持消息的 `id` 一致，这样在 `clientB` 做去重处理就可以解决这个问题。

上面讲的都是基于 `clientB` 在线的情况，但是现实的情况中 `clientB` 往往不可能一直保持在线，因此我们对于离线消息的处理也十分重要

离线消息

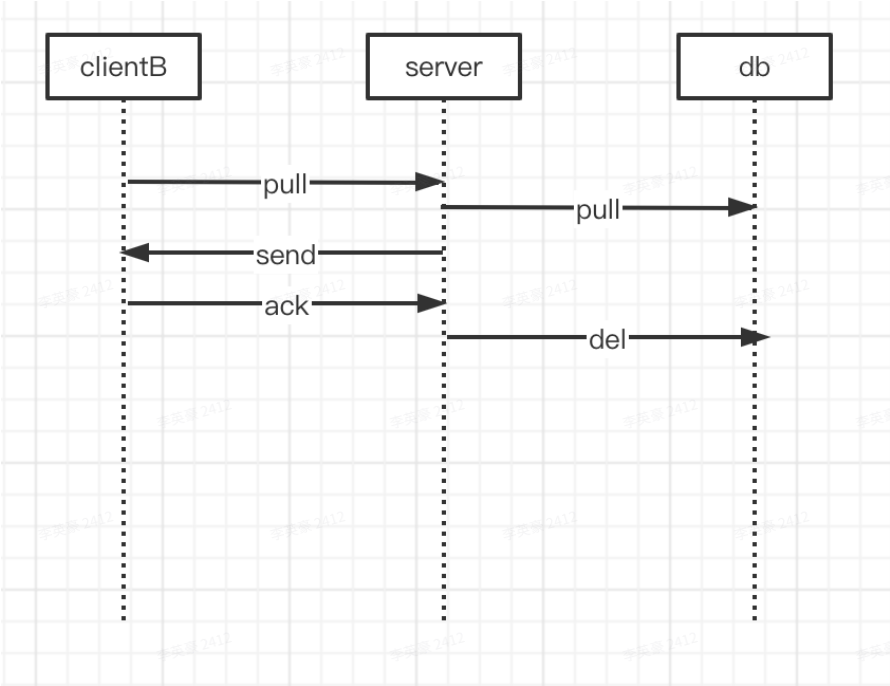
离线消息发送流程：

1. `clientA` 发送消息
2. server回复 `S-ack-to-A` 给 `clientA` ，同时查看 `clientB` 的在线状态，如果 `offline` ，则伪造 `S-ack2-toA` 返回给clientA，同时把消息存储数据库，推送触达用户
3. `clientB` 上线时拉取历史消息， `server` 从数据库拉取消息给 `clientB` 发送push同时删除数据库的数据



以上流程存在一个问题，也就是从 `db` 拉完数据之后服务端推送 `push` 给 `clientB` ，这时 `server` 会从 `db` 删除该消息，这种情况下假如因为网络状况服务端给 `clientB` `push` 的时候丢包了，这种情况下消息就丢失了，为了避免这一种问题，其实还是需要用ACK这种策略,大致如下：

1. clientB向服务端拉取一页消息
2. 服务端向db请求消息后在给到客户端
3. 客户端告诉服务端消息拉取成功
4. 服务端删除消息



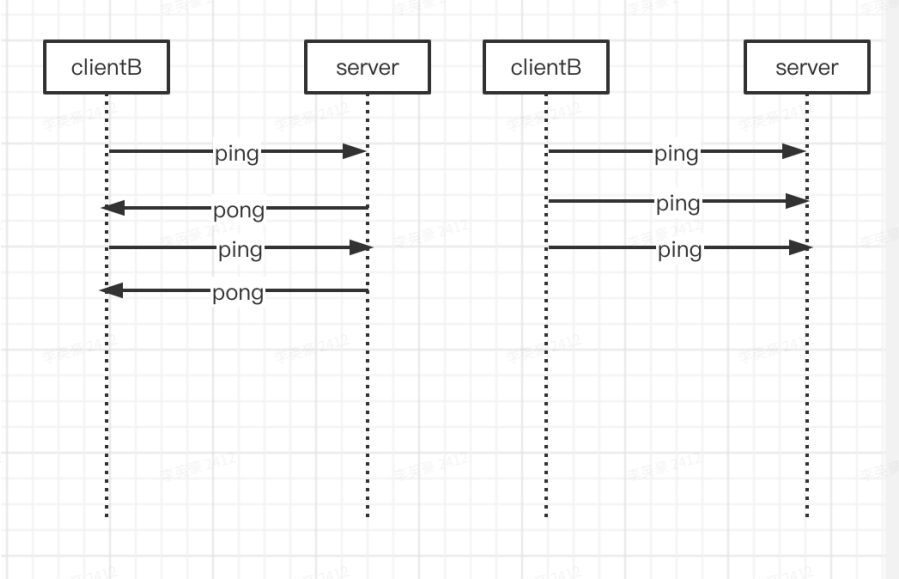
离线消息的流程大概就是上面说的那样，除了离线存储在即时通讯中还有一个很重要的东西就是触达，它到底是什么？

即时触达

即时触达即能够快速的通知到用户，在IM中就是让收消息的一方快速收到明确的通知，在实际的情况中一般分为一下三种情况：

1. app在前台 —— 不触达
2. app在后台活跃中 —— 触达
3. app不活跃 —— 触达

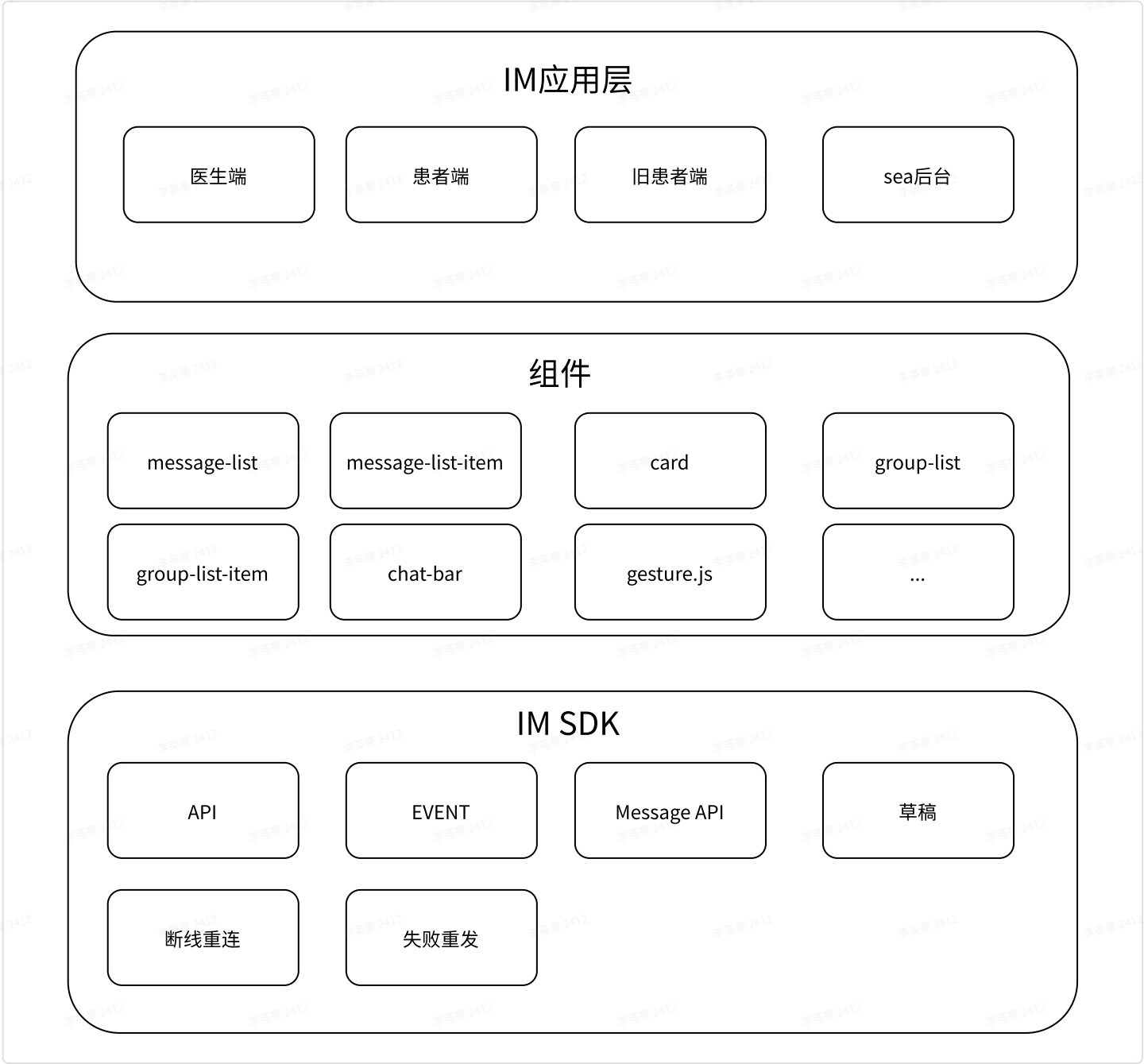
一般在socket建立的时候会在一定的时间内发送心跳包，保持socket的持久化链接同时可以判断是app是否不活跃，类似于下面的流程



客户端在一定时间内收不到服务端的回应，会认为socket断掉了，这个时候一般会进行一定次数的重连尝试。

互联网医院的IM

整体架构



常量

```
export const CONTENT_TYPE = {
  /**
   * 文本消息
   */
  TEXT: 1,
  /**
   * 自定义卡片
   */
  CUSTOM: 3,
  /**
   * 图片消息
   */
  IMAGE: 4,
```

```
  /**
   * 时间
   */
  TIME: 5,
  /**
   * AT消息
   */
  AT: 8,
  /**
   * 系统
   */
  SYSTEM: 9,
  /**
   * 语音消息
   */
  AUDIO: 10,
  /**
   * 撤回
   */
  REVOKE: 11
}

export const MESSAGE_TYPE = {
  /**
   * 心跳包
   */
  PING: 40,
  /**
   * 群聊
   */
  GROUP: 20,
  /**
   * 确认消息
   */
  ACK: 50,
  /**
   * 心跳返回
   */
  PONG: 80,
  /**
   * 已读回执
   */
  RECEIPT: 90,
  /**
   * 已读回执返回值
   */
  RECEIPT_RES: 100,
```

```
61    /**
62     * 群成员变更
63     */
64    GROUP_MEMBER_CHANGE: 110
65  }
```

IM SDK

从上图可以看出IM SDK集成了所有通讯层的方法、内部逻辑，主要涵盖以下几个核心功能

1. websocket链接以及断线重连
2. 消息发送的失败重发
3. 拉取历史消息、诊室列表、诊室成员信息等api
4. 消息体上的API 比如：撤回、重发等
5. 集成的事件 SEND_MESSAGE、MESSAGE_REVOKED、MESSAGE_RECEIVED

那么接下来我们去看它到底做了什么

websocket连接以及断线重连

我们上面说过了为了保持长链接的持久化链接，一般我们是需要发送心跳包的去维持链接的。简易流程如下：

1. 初始化或者收到“pong”时重置心跳计数次数
2. 心跳计数次数>=3 判定连接断开，需要重连

伪代码：

```
// 断线重连
if (this.webSocket.readyState === WebSocket.CLOSED || socketHeartbeatTimesCount >
  this.startWebSocket().then(() => {
      socketHeartbeatTimer = setTimeout(() => {
        this.pollSocketHeartbeat() // 连接失败后 等待500ms 继续重试
      }, 3000) // 三秒一次重新连接
  return
})
// 发送心跳包
this.sendMessage({ messageType: MESSAGE_TYPE.PING, sessionId: this.state.currentS
// 自增+1
socketHeartbeatTimesCount += 1

// 收到心跳 “pong”
socketHeartbeatTimesCount = 0
```

消息可靠发送和失败机制

消息的可靠发送和失败重发机制我们在上面也简单介绍过了，我们这边的流程如下：

1. 调用message api 创建消息带有唯一key，发送该消息，初始状态为loading
2. 以key为键值维护一个map对象，给该条消息设置个定时器，在一定时间内没有收到ACK（也就是定时器被触发），就认定该消息发送失败，改变消息状态
3. 收到该消息的ACK就移除该消息，并修改消息状态为成功

伪代码：

```
// 发送消息
sendMessage(msgData, option = {
  pushFlag: true // 是否通过im在历史消息追加该条消息
}) {
  return new Promise((resolve, reject) => {
    this.pushMsgDataToHistoryMessage(msgData, option)
    // 记录该消息的状态
    this.recordSocketSendStatus(msgData, option, resolve, reject)
    this.webSocket.send(JSON.stringify(msgData))
  })
}
// 记录消息
recordSocketSendStatus(msgData, option, resolve, reject) {
  // 创建一个map对象，存放该promise的resolve和reject
  this.activeMsgDataStatus.promiseMap[msgData.messageKey] = {
    resolve,
    reject,
    timer: null
  }
  // 错误消息队列是否存在该消息
  const isAlreadySend = this.activeMsgDataStatus.errorStash.findIndex(item => ite
  if (isAlreadySend === -1) { // 如果没有添加过
    this.activeMsgDataStatus.errorStash.push(msgData)
  }
  // 创建个定时器，该定时器被触发说明没有收到ack，则把状态设置为error
  this.activeMsgDataStatus.promiseMap[msgData.messageKey].timer = setTimeout(() =
    this.activeMsgDataStatus.promiseMap[msgData.messageKey].reject()
    const target = findRight(this.state.historyMessage, item => item.messageKey =
    rawVue.set(target, 'error', true)
    delete this.activeMsgDataStatus.promiseMap[msgData.messageKey]
  }, 5000)
}

```

```
// 收到ack
[MESSAGE_TYPE.ACK]: () => { // 消息确认
  // 确认消息是否送达
  const index = findIndexRight(this.state.historyMessage, item => item.messageKey
  const ackMsgItem = this.state.historyMessage[index]
  if (ackMsgItem) {
    rawVue.set(ackMsgItem, 'messageId', message.messageId)
    rawVue.set(ackMsgItem, 'createTime', message.createTime)
    rawVue.set(ackMsgItem, 'ready', true)
    // 自己发送成功的消息通知其他群成员自己已读
    this.createReceiptMessage({ readMessageId: message.messageId }).sendMessage()
  }
  // 处理消息的发送状态
  const promiseItem = this.activeMsgDataStatus.promiseMap[ackMsgItem.messageKey]
  const ackMsgItemErrorStashIdx = this.activeMsgDataStatus.errorStash.findIndex(i
  this.activeMsgDataStatus.errorStash.splice(ackMsgItemErrorStashIdx, 1) // 成功后
  if (promiseItem) {
    promiseItem.resolve()
    clearTimeout(promiseItem.timer)
    delete this.activeMsgDataStatus.promiseMap[ackMsgItem.messageKey]
  }
}
```

事件机制

事件机制比较简单，自己实现了一套简易的发布订阅模式，类似于js中的addEventListener dispatchEvent。支持的event如下：

1. MESSAGE_RECEIVED 收到新消息会触发
2. MESSAGE_REVOKED 消息被撤回会触发
3. SEND_MESSAGE 发送消息时会触发

设计这个东西的原因是一开始想要做成中间件的机制，但是由于时间的问题最终是简版的实现，主要解决一些不方便在组件层处理的逻辑。

```
// IMEvent class
export class IMEvent {
  constructor() {
    this.listeners = {}
  }
  addEventListener(type, callback) {
    if (!(type in this.listeners)) {
      this.listeners[type] = []
    }
    this.listeners[type].push(callback)
```

```
  }
  removeEventListener(type, callback) {
    if (!(type in this.listeners)) {
      return
    }
    const stack = this.listeners[type]
    // eslint-disable-next-line no-plusplus
    for (let i = 0, l = stack.length; i < l; i++) {
      if (stack[i] === callback) {
        stack.splice(i, 1)
        // eslint-disable-next-line consistent-return
        return this.removeEventListener(type, callback)
      }
    }
  }
  dispatchEvent(event) {
    if (!(event.type in this.listeners)) {
      return
    }
    const stack = this.listeners[event.type]
    // eslint-disable-next-line no-plusplus
    for (let i = 0, l = stack.length; i < l; i++) {
      stack[i].call(this, event)
    }
  }
}
```

Message API

IM内部实现了一批关于Message的API

1. createTextMessage
2. createATMessage
3. createAudioMessage
4. createImageMessage
5. createTimeMessage

并给每一条message上包装了一些原型方法：

1. resend - 重新发送
2. send - 发送
3. revoke - 撤回

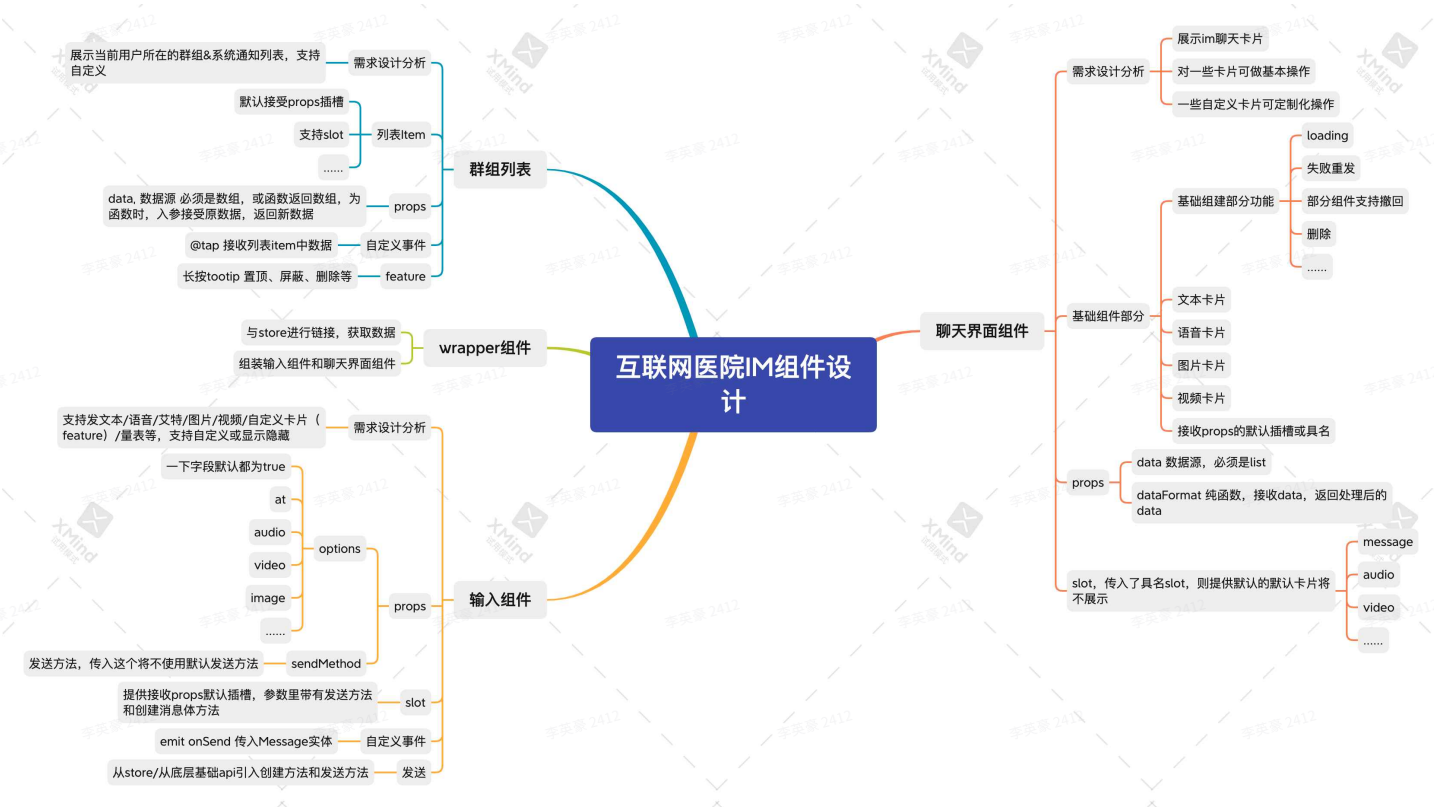
这样做的主要目的是为了不与业务有太高得耦合性，在 `message` 组件上只要提供了这些 `api` 就可以直接使用这些方法。假如之后有消息引用、消息收藏、翻译等功能都可以直接在im层实现这个接口功

能，组件内部就可以直接使用

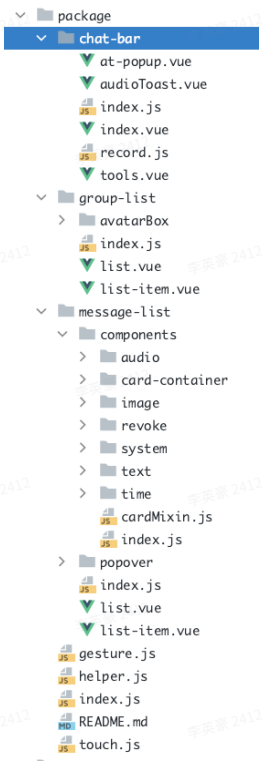
到这里SDK部分核心能力大致已经完事儿了，接下来我们看下UI组件层的设计

IM UI 组件

最初版的设计：



目录结构



1. chat-bar im内底部的输入框和功能集合

2. group-list 会话列表
    a. list 回话列表
    b. list-item 单个会话
3. message-list 消息列表
    a. list 诊室内消息列表
    b. list-item 单个消息，支持default slot
    c. components 提供的基础消息组件集合
4. popover 长按操作弹窗
5. guester 手势库
6. helper 工具集
7. touch 自定义指令，链接手势库和UI组件

message-list

伪代码如下：

```
export default {
  name: 'MessageList',
  props: {...},
  data() {...},
  computed: {...},
  mounted() {
    const slots = this.$slots.default || []
    slots.forEach((slot) => {
      const slotProps = getPropsFromVnode(slot)
      this.customCardTypes = [...this.customCardTypes, slotProps.type]
    })
    ...
  },
  beforeUpdate() {...},
  methods: {...},
  render(h) {
    ...
    const list = this.convertedData.map((message) => {
      // 需要自定义聊天card的数据
      if (this.customCardTypes.includes(message.contentType)) {
        // eslint-disable-next-line no-unused-vars
        return Object.entries(this.$slots.default).map(([name, slot]) => {
          const slotProps = getPropsFromVnode(slot)
          if (slotProps.type !== message.contentType) {
            return null
```

```
          }
          const newNode = deepCloneNode([slot], h)
          setPropsToVnode(newNode[0], { message, type: message.contentType })
          return newNode
        })
      }
      return (
        <MessageListItem
          message={message}
          key={message.messageId}
          id={`MessageListItem${message.messageId}`}
          type={message.contentType}
          onClickAvatar={m => this.$emit('click-avatar', m)}
          onClickImage={() => preview(message)}
        />
      )
    })
    return (
      <div class="message-list" onScroll={ev => this.scroll(ev)} ref="messageList
        {this.loadNexting && (
          <div class="loading">
            { <dpui-loading size="24px">加载中...</dpui-loading> }
          </div>
        )}
        {list}
      </div>
    )
  }
}

// message-list-item 伪代码如下
<div class="message-list-item">
  <slot v-if="$scopedSlots.default" :message="message" />
  <div v-else>
    <component
      :is="cardType"
      :message="message"
      @click-avatar="$emit('clickAvatar', message)"
      @click-image="$emit('clickImage', message)"
    />
  </div>
</div>
```

1. 需要message-list默认情况下根据消息type展示基础组件
2. 需要用户可以根据指定的type自定义组件

a. 从default中取出message参数进行客制化逻辑渲染

```
<message-list
  ref="msgNode"
  :data="historyMessage"
  :data-format="dataFormat"
  :loading="loading"
  :load-next-func="loadMore"
  @click-avatar="clickAvatar"
>
  <!-- 自定义卡片类型为3 -->
  <message-list-item :type="3">
    <template v-slot="{message}">
      <card-container
        v-if="getCustomCardType(message) !== 6 && getCustomCardType(message) !==
        :avatar="message.headImageUrl || getDefaultAvatarByRole(message.role)"
        :nickname="message.userName"
        @click-avatar="clickAvatar(message)"
      >
        <!-- 医生卡片类型为4 -->
        <doctor-info
          v-if="getCustomCardType(message) === 4"
          slot="content"
          :message="message"
          @click.native="showDoctor(message)"
        />
      </card-container>
      <e-prescription
        v-if="getCustomCardType(message) === 0"
        slot="content"
        :message="message"
      />
    </template>
  </message-list-item>
  <message-list-item :type="7">
    <template v-slot="{message}">
      <card-container
        :avatar="message.headImageUrl || getDefaultAvatarByRole(message.role)"
        :nickname="message.userName"
        :self="message.self"
        @click-avatar="clickAvatar(message)"
      >
        <!-- 患者主诉信息 -->
        <chief-complaint
          slot="content"
          :message="message"
```

```
45          />
46        </card-container>
47      </template>
48    </message-list-item>
49  </message-list>
```

经验/踩坑

1. 诊室内任何消息的改动会影响各个端，在我们这里体现在C/B，需要综合考虑在不同端的展示情况
    a. 比如针对于图片。一般的图片进入诊室需要把宽高给定出来，方便滚动到底部时不会因为图片后置加载出来宽高而影响没有滚动到最新的消息
2. 诊室内任何消息操作都是需要广播到其他端
    a. 诊室成员变更
    b. 撤回消息
3. 新增的消息类型要考虑到发送方和接收方的ui展示、PC端的展示
4. 消息宁可多也不能少，多了可以去重展示，少了就导致消息丢失