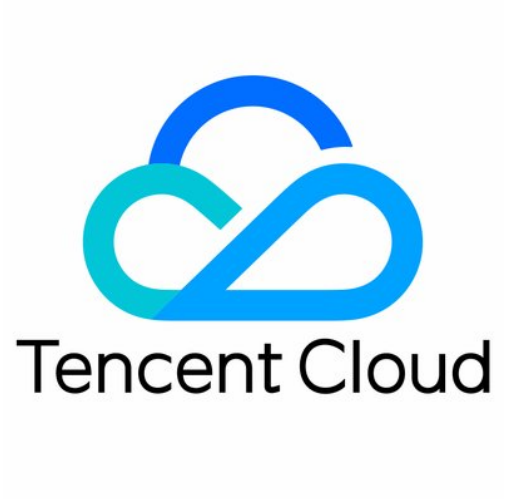
Instant Messaging

Best Practice

Product Documentation



Contents

Best Practice
Enabling the Livestreaming Group Feature

Last updated : 2020-09-08 10:08:59

IM livestreaming groups (AVChatRooms) have the following features:

- **Support interactive livestreaming scenarios with unlimited group members.**
- **Support content filtering to identify harmful content related to pornography, politics, and sensitive words to meet regulatory requirements.**
- Support pushing messages (group system notifications) to all online members.
- Allow users to receive messages as guests without logging in using the web client or WeChat Mini Program.
- Allow users to join a group without admin approval.

Note :

This document uses the SDK for the web client and WeChat Mini Program as an example. The implementation processes of other SDKs are the same with slightly different operations.

Use Cases

On-screen comments for livestreaming

A livestreaming group (AVChatRoom) supports various message types, including on-screen comments, gifts, and likes. It can easily build a good chatting and interactive experience for livestreaming. It also supports on-screen comment moderation to protect your livestreams from sensitive words.

Influencer marketing

A livestreaming group (AVChatRoom), when combined with business livestreaming, supports specific message types, such as likes, inquiries, and vouchers, helping you monetize your traffic.

Teaching whiteboard

A livestreaming group (AVChatRoom) provides capabilities, such as online classrooms, text messages, and pen motion sensing to support teaching scenarios, such as communication between teachers and students, pen motion saving, large classes, and small classes.

Use limits

- Recalling messages is not supported.
- The group owner cannot directly quit the group and can only achieve the same result by disbanding the group.
- Group members cannot be removed.

Related Documentation

- Group Management
- Group System
- SDK Download
- Changelog (Web and Mini Programs)
- SDK Manual
- SDK Integration (Web & Mini Programs)
- Initialization (Web & Mini Programs)
- Sending and Receiving Messages (Web & Mini Programs)
- Group Management (Web & Mini Programs)

Instructions

Step 1: create an app

1. Log in to the IM console.

> **Note :**
>
> If you already have an app, take note of the SDKAppID and go to Step 2.
> A Tencent Cloud account supports a maximum of 100 IM apps. If you have reached that limit, disable and delete an unwanted app before creating a new one. **Once an app is deleted, all the data and services associated with the SDKAppID are removed and cannot be recovered, so proceed with caution.**

2. Click **Add Application**.
3. In the **Create Application** dialog box, enter a name for your app and click **OK**. After the app is created, you can view the status, version, SDKAppID, creation time, and expiration time of the app on the overview page of the console.

4. Take note of the SDKAppID.

Step 2: create a livestreaming group (AVChatRoom)

You can create a group in the console or by calling Creating a Group. This document creates a group in the console.

1. Log in to the IM console and click the target app card.
2. In the left sidebar, select **Group Management** and click **Add Group**.
3. Enter a name for the group and the group owner ID (optional). Select **AVChatRoom** for **Group Type**.
4. Click **OK**. After the group is created, take note of the **Group ID**. `@TGS#aC72FIKG3` is used as an example here.

Step 3: integrate the SDK

You can integrate the SDK using NPM or Script, and we recommend NPM. The example in this document uses NPM integration.

- Web project

```
// Web project
npm install tim-js-sdk --save-dev
```

- Mini Program project

```
// WeChat Mini Program project
npm install tim-wx-sdk --save-dev
```

Note :

If a problem occurs during dependency synchronization, change the NPM source and try again.

```
// Change the CNPM source.
npm config set registry http://r.cnpmjs.org/
```

Step 4: create an SDK instance

```
// Create an SDK instance. The TIM.create() method returns the same instance for the same SDKAppID.
let options = {
SDKAppID: 0 // Replace 0 with the SDKAppID of your IM application during integration.
}
```

```
let tim = TIM.create(options) // The SDK instance is usually represented by tim.
// Set the SDK logging level. For more information, see setLogLevel Description.
tim.setLogLevel(0) // Normal level. We recommend that you use this level during integration as it covers more logs.

tim.on(TIM.EVENT.SDK_READY, function (event) {
// The access side can call APIs that require authentication, such as sendMessage, only after the SDK is ready. Otherwise, the calls will fail.
// event.name - TIM.EVENT.SDK_READY
})

tim.on(TIM.EVENT.MESSAGE_RECEIVED, function(event) {
// A newly pushed one-to-one message, group message, group prompt, or group system notification is received. You can traverse `event.data` to obtain the message list and render it to the UI.
// event.name - TIM.EVENT.MESSAGE_RECEIVED
// event.data - an array that stores Message objects - [Message]
const length = event.data.length
let message
for (let i = 0; i < length; i++) {
// For more information on the data structure of Message instances, see Message.
// Pay particular attention to the type and payload properties.
// Starting from SDK v2.6.0, the nick (nickname) and avatar (profile photo URL) properties are added for group chat messages and group prompts for events, such as users joining and quitting live streaming groups. This allows the access side to provide a better display experience.
// To do this, you must first set your own nick (nickname) and avatar (profile photo URL) by calling updateMyProfile. For more information, see updateMyProfile Description.
message = event.data[i]
switch (message.type) {
case TIM.TYPES.MSG_TEXT:
// A text message is received.
this._handleTextMsg(message)
break
case TIM.TYPES.MSG_CUSTOM:
// A custom message is received.
this._handleCustomMsg(message)
break
case TIM.TYPES.MSG_GRP_TIP:
// A group prompt about an event (such as when a user joins the group or a member quits the group) is received.
this._handleGroupTip(message)
break
case TIM.TYPES.MSG_GRP_SYS_NOTICE:
// A group system notification is received. For more information on group system notifications sent using RESTful APIs, see Sending System Messages in a Group.
this._handleGroupSystemNotice(message)
break
default:
break
```

```
}
}
})

_handleTextMsg(message) {
// For more information on the data structure, see TextPayload Description.
console.log(message.payload.text) // Text message content
}

_handleCustomMsg(message) {
// For more information on the data structure, see CustomPayload Description.
console.log(message.payload)
}

_handleGroupTip(message) {
// For more information on the data structure, see GroupTipPayload Description.
switch (message.payload.operationType) {
case TIM.TYPES.GRP_TIP_MBR_JOIN: // A user joins the group.
break
case TIM.TYPES.GRP_TIP_MBR_QUIT: // A member quits the group.
break
case TIM.TYPES.GRP_TIP_MBR_KICKED_OUT: // A member is removed from the group.
break
case TIM.TYPES.GRP_TIP_MBR_SET_ADMIN: // A member is set as an admin.
break
case TIM.TYPES.GRP_TIP_MBR_CANCELED_ADMIN: // A member's admin role is canceled.
break
case TIM.TYPES.GRP_TIP_GRP_PROFILE_UPDATED: // The group profile is modified.
//Starting from v2.6.0, group custom fields can be modified.
// message.payload.newGroupProfile.groupCustomField
break
case TIM.TYPES.GRP_TIP_MBR_PROFILE_UPDATED: // The group member profile is modified. For example,
the member is muted.
break
default:
break
}
}

_handleGroupSystemNotice(message) {
// For more information on the data structure, see GroupSystemNoticePayload Description.
console.log(message.payload.userDefinedField) // User-defined field. When sending a group system
notification using a RESTful API, you can get the content of the custom notification in this prop
erty value.
// For more information on how to use a RESTful API to send group system notifications, see Sendi
ng System Messages in a Group.
}
```

Step 5: log in to the SDK

```
let promise = tim.login({userID: 'your userID', userSig: 'your userSig'});
promise.then(function(imResponse) {
console.log(imResponse.data); // Login successful.
}).catch(function(imError) {
console.warn('login error:', imError); // Information about the login failure.
});
```

Step 6: set your nickname and profile photo

Starting from SDK v2.6.2, the nick (nickname) and avatar (profile photo URL) properties are added for group chat messages and group prompts for events, such as joining and quitting groups in livestreaming groups. You can call updateMyProfile to set your nickname and profile photo.

```
// Starting from SDK v2.6.0, the nick (nickname) and avatar (profile photo URL) properties are ad
ded for group chat messages and group prompts for events, such as joining and quitting groups in
livestreaming groups. This allows the access side to provide a better display experience. To do t
his, you must first set your own profile by calling updateMyProfile.
// Modify your personal standard profile.
let promise = tim.updateMyProfile({
nick: 'My nickname',
avatar: 'http(s)://url/to/image.jpg'
});
promise.then(function(imResponse) {
console.log(imResponse.data); // The profile was updated.
}).catch(function(imError) {
console.warn('updateMyProfile error:', imError); // Information on the failure to update your pro
file.
});
```

Step 7: join a group

```
// An anonymous user joins the group (no login is required and the user will only receive message
s after joining the group).
let promise = tim.joinGroup({ groupID: 'avchatroom_groupID'});
promise.then(function(imResponse) {
switch (imResponse.data.status) {
case TIM.TYPES.JOIN_STATUS_WAIT_APPROVAL: // Waiting for the admin's approval.
break
case TIM.TYPES.JOIN_STATUS_SUCCESS: // Joined the group successfully.
console.log(imResponse.data.group) // The profile of the group.
break
case TIM.TYPES.JOIN_STATUS_ALREADY_IN_GROUP: // The user is already in the group.
break
```

```
default:
break
}
}).catch(function(imError){
console.warn('joinGroup error:', imError) // Information on the failure to join the group.
});
```

Step 8: create a message instance and send a message

This document uses sending a text message as an example.

```
// Send a text message; same for the web client and WeChat Mini Program.
// 1. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createTextMessage({
to: 'avchatroom_groupID',
conversationType: TIM.TYPES.CONV_GROUP,
// Group chat message priority (supported by v2.4.2 and later). If messages in a group exceed the frequency limit, the backend delivers high-priority messages first. For more information, please see "Message Priority and Frequency Control" in Group Chat Messages.
// Valid values: TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL (default), TIM.TYPES.MSG_PRIORITY_LOW, TIM.TYPES.MSG_PRIORITY_LOWEST
priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
payload: {
text: 'Hello world!'
}
})
// 2. Send a message.
let promise = tim.sendMessage(message)
promise.then(function(imResponse) {
// Sent successfully.
console.log(imResponse)
}).catch(function(imError) {
// Failed to send.
console.warn('sendMessage error:', imError)
})
```

FAQs

1. If I send a message in which `Message.nick` and `Message.avatar` are both empty, how can the message be processed to normally display the nickname and profile photo on the screen?

You can call getMyProfile to obtain your nickname and profile photo.

2. How can I mute a group member in a livestreaming group?

You can use a custom message to mute a specific group member. The custom message must contain the Members_Account of the group member to be muted and the muting time. Send the custom message to the business backend by calling the Callback Before Delivering Group Messages. The business backend will call Muting and Unmuting Group Members to mute the group member.

3. How can I remove a group member from a livestreaming group?

You can use a custom message to remove a specific group member. The custom message must contain the Members_Account of the group member to be removed. Set the priority of the custom message to High to prevent the message from being discarded by the backend due to the message sending frequency limit of 40 messages per second. After the SDK receives the message, it calls the kickGroupMember API to remove the group member from the group.

4. A group member quits a livestreaming group on the Android, iOS, or PC client when the API for quitting a group is called on the WeChat Mini Program or web client. However, when the API for quitting a group is called on the Android, iOS, or PC client, the group member does not quit the livestreaming group on the WeChat Mini Program or web client. Why is this the case?

The WeChat Mini Program and web client allow users to join livestreaming groups as guests. When the API for quitting a group is called on the Android, iOS, or PC client, the WeChat Mini Program or web client will not proactively trigger the quit group operation.

- To ensure synchronized group quitting on all clients, configure the Callback After a Group Member Drops Out and determine the quit platform based on the OptPlatform field. When the quit platform is Android, iOS, or PC, call Send a One-to-One Message to send a custom message to the group member who quits the group. The frontend screens the conversation and does not display it on the UI. After the WeChat Mini Program or web client receives the message, it calls the quitGroup API.
- To ensure independent group quitting on different clients, configure Callback After a Group Member Drops Out and determine the quit platform based on the OptPlatform field. Call Send a One-to-One Message to send a custom message to the group member who quits the group. The frontend screens the conversation and does not display it on the UI. After a non-quit platform receives the message, it calls the joinGroup API to join the group. To prevent multiple system notifications for joining and quitting a group, you can submit a ticket to disable system notifications for joining and quitting a group.

5. Why are messages lost?

Messages can be lost due to the following conditions:

- Livestreaming groups have a message sending frequency limit of 40 messages per second. You can determine whether a message is discarded due to the frequency limit based on the callbacks before and after delivering group messages. If a message receives the callback before delivering group messages but does not receive the callback after delivering group messages, the message is discarded due to the frequency limit.
- When a group member quits a livestreaming group from the WeChat Mini Program or web client, the member may also quit on the Android, iOS, or PC client. For more information, see FAQ 4.
- If the WeChat Mini Program or web client encounters an exception, check whether your SDK version is earlier than v2.7.6. If yes, upgrade your SDK to the latest version.

If the fault persists, submit a ticket to contact us.

6. How can I compile statistics on likes and follows?

Customize the like or follow message type. When a user clicks the like or follow icon on the frontend to deliver a custom message, send the like or follow message to the business side by calling the Callback Before Delivering Group Messages. The business side counts the number of received like and follow messages and updates the data to the group profile field every 3 to 5 seconds by calling Modify Basic Group Profiles. The SDK calls the getGroupsInfo API to count the number of like or follow messages.

7. How can I properly set message priorities?

To prevent important messages from being discarded, a livestreaming group provides three message priorities for all messages. The SDK preferentially obtains high priority messages. We recommend that you set the priorities of custom messages as follows:

- High: red packets, gifts, and messages for removing group members
- Normal: common text messages
- Low: likes and follows

8. Are there any open-source livestreaming components that can be directly used to watch videos and chat?

Yes. The code is also open-source. For more information, see Tencent Cloud TWebLive.

Guide to the use of AVChatRoom (Web & Mini Program)

Last updated : 2020-05-14 11:45:21

An audio-video chat room (AVChatRoom) has the following characteristics:

- **Suitable for interactive live video broadcasting scenarios with unlimited group members**.
- **Supports content filtering to identify harmful content related to pornography, politics, and sensitive words to meet regulatory requirements.
- Supports pushing messages (group system notifications) to all online members.
- Web and WeChat Mini Program clients allow users to receive messages as guests without logging in.
- Users can enter the group directly with no admin approval required.

Use Cases

On-screen comments for live video broadcasting

AVChatRoom makes it easy to build good chatting and interactive experience in live video broadcasting by supporting various message types including on-screen comments, gifts, and likes. The content moderation capability can protect your livestreams from sensitive words in on-screen comments.

Influencer marketing

AVChatRoom, when used with business live streaming, supports messages such as likes, inquiries, and vouchers, helping you turn traffic into profit.

Teaching whiteboard

AVChatRoom provides capabilities such as online classroom, text messages, and pen motion, assisting teaching scenarios such as communication between teachers and students, saving pen motion, large classes, and small classes.

Use limits

- Recalling messages is not supported.
- Group owner cannot directly quit the group and can only do so by disbanding the group.

- Group members cannot be removed.

Related Documentation

- Group Management
- Group System
- SDK Download
- Changelog (Web and Mini Programs)
- SDK Manual
- SDK Integration (Web & Mini Programs)
- Initialization (Web & Mini Programs)
- Login (Web & Mini Programs)
- Sending and Receiving Messages (Web & Mini Programs)
- Group Management (Web & Mini Programs)

Instructions

Step 1: Create an app

1. Log in to the Tencent Cloud IM Console.

> If you already have an app, take note of the SDKAppID and go to step 2.
> A Tencent Cloud account supports a maximum of 100 IM apps. If 100 IM apps have been created, you can create a new app after disabling and deleting an unwanted app. **After an app is deleted, all the data and services associated with the SDKAppID cannot be recovered. Proceed with caution.**

2. Click **Add a new app**.
3. In the **Create App** dialog box, enter a name for your app and click **OK**. After the app is created, you can view the status, service version, SDKAppID, creation time and expiration time of the app on the overview page of the console.
4. Take note of the SDKAppID.

Step 2: Create an audio-video chat room (AVChatRoom)

You can create a group from the console or by calling the API to create a group. This document creates a group from the console.

1. Log in to the Instant Messaging Console, and then click the target app card.
2. In the left sidebar, select **Group Management** and click **Add Group**.
3. Enter a name for the group and the group owner ID (optional). For **Group type**, select **Audio-video chat room**.
4. Click **OK**. After the group is created, take note of the **Group ID** (`@TGS#aC72FIKG3` is used as an example here).

Step 3: integrate the SDK

You can integrate the SDK using NPM or Script, though NPM is recommended. This document uses NPM integration as an example.

- Web project

```
// Web project
npm install tim-js-sdk --save-dev
```

- Mini Program project

```
// WeChat Mini Program project
npm install tim-wx-sdk --save-dev
```

If a problem occurs during dependency synchronization, change npm source and try again.

```
// Change cnpm source
npm config set registry http://r.cnpmjs.org/
```

Step 4: Create an SDK instance

```
// Create an SDK instance. The `TIM.create()` method returns the same instance for the same `SDKAppI
let options = {
  SDKAppID: 0 // Replace 0 with the SDKAppID of your IM app when connecting
}
let tim = TIM.create(options) // SDK instance is usually represented by tim
// Set SDK logging level. For a detailed description of each level, please see setLogLevel.
tim.setLogLevel(0) // Normal level. We recommend using this level during connection as it covers lar

tim.on(TIM.EVENT.SDK_READY, function (event) {
  // The access side can call APIs that require authentication (for example, sendMessage) only after
  // event.name - TIM.EVENT.SDK_READY
})
```

```
tim.on(TIM.EVENT.MESSAGE_RECEIVED, function(event) {
  // Received new one-to-one, group, group tips, and group system notification messages that are pus
  // event.name - TIM.EVENT.MESSAGE_RECEIVED
  // event.data - an array that stores Message objects - [Message]
  const length = event.data.length
  let message
  for (let i = 0; i < length; i++) {
    // For more information on the data structure of Message instances, see Message.
    // Note particularly the type and payload properties.
    // Starting from v2.6.0, the nick (nickname) and avatar (profile photo URL) properties are added
    // To do this, you must first set your own nick (nickname) and avatar (profile photo URL) by cal
    message = event.data[i]
    switch (message.type) {
      case TIM.TYPES.MSG_TEXT:
        // A text message is received.
        this._handleTextMsg(message)
        break
      case TIM.TYPES.MSG_CUSTOM
        // A custom message is received.
        this._handleCustomMsg(message)
        break
      case TIM.TYPES.MSG_GRP_TIP:
        // A group tips message (for example, joining group and quitting group) is received.
        this._handleGroupTip(message)
        break
      case TIM.TYPES.MSG_GRP_SYS_NOTICE:
        // A group system notification is received. For group system notifications sent via RESTful
        this._handleGroupSystemNotice(message)
        break
      default:
         break
    }
  }
})

_handleTextMsg(message) {
  // For more information on the data structure, see the descriptions of API TextPayload.
  console.log(message.payload.text) // Text message content
}

_handleCustomMsg(message) {
  // For more information on the data structure, see the descriptions of API CustomPayload.
  console.log(message.payload)
}

_handleGroupTip(message) {
  // For more information on the data structure, see the descriptions of API GroupTipPayload.
```

```
  switch (message.payload.operationType) {
    case TIM.TYPES.GRP_TIP_MBR_JOIN: // A member joins the group.
      break
    case TIM.TYPES.GRP_TIP_MBR_QUIT: // A member quits the group.
      break
    case TIM.TYPES.GRP_TIP_MBR_KICKED_OUT: // A member is kicked out of the group.
      break
    case TIM.TYPES.GRP_TIP_MBR_SET_ADMIN: // A member is set as admin.
      break
    case TIM.TYPES.GRP_TIP_MBR_CANCELED_ADMIN: // The admin role of a member is revoked.
      break
    case TIM.TYPES.GRP_TIP_GRP_PROFILE_UPDATED: // Group profile is modified.
      // Modifying group custom fields is supported by v2.6.0 and higher.
      // message.payload.newGroupProfile.groupCustomField
      break
    case TIM.TYPES.GRP_TIP_MBR_PROFILE_UPDATED: // Group member profile is modified. For example, a
      break
    default:
      break
  }
}

_handleGroupSystemNotice(message) {
  // For more information on the data structure, see the descriptions of API GroupSystemNoticePayloa
  console.log(message.payload.userDefinedField) // User-defined field. When sending a group system r
  // To send group system notifications via RESTful APIs, see the API for sending system notificatio
}
```

Step 5: Join a group

```
// An anonymous user joins the group (no login is required and the user will only receive message
s after joining the group).
let promise = tim.joinGroup({ groupID: 'avchatroom_groupID'});
promise.then(function(imResponse) {
switch (imResponse.data.status) {
case TIM.TYPES.JOIN_STATUS_WAIT_APPROVAL: // Waiting for the admin's approval.
break
case TIM.TYPES.JOIN_STATUS_SUCCESS: // Joined the group successfully.
console.log(imResponse.data.group) // The profile of the group.
break
case TIM.TYPES.JOIN_STATUS_ALREADY_IN_GROUP: // The user is already in the group.
break
default:
break
}
}).catch(function(imError){
```

```
console.warn('joinGroup error:', imError) // Information on the failure in joining the group.
});
```

Step 6: Log in to the SDK

```
let promise = tim.login({userID: 'your userID', userSig: 'your userSig'});
promise.then(function(imResponse) {
console.log(imResponse.data); // Login succeeded.
}).catch(function(imError) {
console.warn('login error:', imError); // Information about login failure.
});
```

Step 7: Create a message instance and send a message

This document takes sending a text message as an example.

```
// Send a text message; same for web applications and Mini Programs.
// 1. Create a message instance. The instance returned by the API can be displayed on the screen.
let message = tim.createTextMessage({
  to: 'avchatroom_groupID',
  conversationType: TIM.TYPES.CONV_GROUP,
  // Message priority applicable to group chats (supported by v2.4.2 and higher). If messages in a g
  // Enumerated values supported: TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORITY_NORMAL (defaul
  priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
  payload: {
    "Text": "hello world"
  }
})

// 2. Send the message.
let promise = tim.sendMessage(message)
promise.then(function(imResponse) {
  // Sent successfully.
  console.log(imResponse)
}).catch(function(imError) {
  // Failed to send
  console.warn('sendMessage error:', imError)
})
```

Muting/Unmuting (Web & Mini Programs)

Last updated : 2020-06-23 15:28:42

You can mute a group member or all group members for a specified period of time. During the muting period, the muted member cannot send messages in the current group. This muting operation is effective only for the current group. During the muting period, even if the muted member quits the group and rejoins it, the muting will remain effective until the muting period is over or the member is unmuted.
This document mainly describes how to mute and unmute group members in web and Mini Program SDKs.

Use Limits

- Group type limits

Work Group (Work or Private of the Earlier Version)	Public Group (Public)	Meeting Group (Meeting or ChatRoom of the Earlier Version)	Audio-Video Chat Room (AVChatRoom)
Not supported	Supported	Supported	Supported

- Member role restrictions

Member Role	Permissions
App admin	App admins are allowed to mute or unmute all members in all groups under the current SDKAppID.
Group owner	The group owner is allowed to mute or unmute admins and common members in the current group.
Group admin	Group admins are allowed to mute or unmute common members in the current group.
Common member	No muting permissions

Directions

Step 1: confirm the operation permission

1. Call the getGroupProfile API to query the type of the current group and confirm whether muting/unmuting is supported.

   > If the type of the group is Private or Work (work group), muting is not supported.

2. Call the getGroupMemberProfile API to query the member role of the specified userID in the current group and confirm whether the user has the permission to perform muting/unmuting.

Step 2: mute/unmute a group member

Muting/Unmuting a single user

App admins, the group owner, or group admins can call the setGroupMemberMuteTime API to mute or unmute a specified member in the specified group. For each call, only 1 member can be muted/unmuted.
The following table lists the request parameters.

Name	Type	Description
groupID	String	Group ID
userID	String	Group Member ID
muteTime	Number	The muting period, in seconds. 0: unmuting.

A sample request is shown as follows:

```
let promise = tim.setGroupMemberMuteTime({
groupID: 'group1',
userID: 'user1',
muteTime: 1000
});
```

Muting/Unmuting all members

> To use this feature, you need to upgrade the SDK to 2.6.2 or higher. At present, for muting and unmuting all members, no group tip message is delivered.

App admins or the group owner can call the updateGroupProfile API to mute/unmute all admins and common members in the specified group.
The following table lists the request parameters.

Name	Type	Description
groupID	String	Group ID
muteAllMembers	Boolean	Set muting. true: mute all members. false: unmute all members.

A sample request is shown as follows:

```
let promise = tim.updateGroupProfile({
groupID: 'group1',
muteAllMembers: true, // true: mute all members. false: unmute all members.
});
promise.then(function(imResponse) {
console.log(imResponse.data.group) // Detailed group profile after modification
}).catch(function(imError) {
console.warn('updateGroupProfile error:', imError); // Information on the failure in modifying th
e group profile
});
```

Step 3: monitor and process the event TIM.EVENT.MESSAGE_RECEIVED

After a group member is muted, the member will receive a group tip indicating that he/she has been muted. You can traverse `event.data` to get relevant data and render it to the interface.

> After relevant muting/unmuting notifications are received, we recommend that you implement the disable/enable input box or input area status.

```
tim.on(TIM.EVENT.MESSAGE_RECEIVED, function(event) {
  // A newly pushed one-to-one message, group message, group tip, or group system notification is re
  // event.name - TIM.EVENT.MESSAGE_RECEIVED
  // event.data - An array that stores the Message objects - [Message]
  const length = event.data.length;
```

```
  let message;
  for (let i = 0; i < length; i++) {
    message = event.data[i];
    switch (message.type) {
      // Mute user A, user A will receive a group tip message indicating that he/she has been muted.
      case TIM.TYPES.MSG_GRP_TIP:
        this._handleGroupTip(message);
        break;
      case TIM.TYPES.MSG_TEXT: // Text message. For more message types, see Message.
        break;
      default:
        break;
    }
  }
});

_handleGroupTip(message) {
  switch (message.payload.operationType) {
    case TIM.TYPES.GRP_TIP_MBR_PROFILE_UPDATED: // Group member profile is modified. For example, a
      const memberList = message.payload.memberList;
      for (let member of memberList) {
        console.log(`${member.userID} muted for ${member.muteTime} seconds`);
      }
      break;
    case TIM.TYPES.GRP_TIP_MBR_JOIN: // A member joins the group
      break;
    case TIM.TYPES.GRP_TIP_MBR_QUIT: // A member quits the group
      break;
    case TIM.TYPES.GRP_TIP_MBR_KICKED_OUT: // A member is kicked out of the group
      break;
    case TIM.TYPES.GRP_TIP_GRP_PROFILE_UPDATED: // Group profile modified
      break;
    default:
      break;
  }
}
```

Step 4: check the muting status

- For SDK 2.6.2 and higher, you can call the getGroupMemberList API to pull the muting stop timestamp (muteUntil) of a group member. Based on its value, you can find out whether the member is muted and the remaining muting time. After the member is unmuted, the following calculation formula for the member is valid: GroupMember.muteUntil * 1000 <= Date.now().

```
// Starting from v2.6.2, the getGroupMemberList API can be used to pull the muting stop timest
amps of group members.
let promise = tim.getGroupMemberList({ groupID: 'group1', count: 30, offset:0 }); // Pull 30 g
```

```
roup members starting from 0
promise.then(function(imResponse) {
console.log(imResponse.data.memberList); // Group member list
for (let groupMember of imResponse.data.memberList) {
if (groupMember.muteUntil * 1000 > Date.now()) {
console.log(`${groupMember.userID} muted`);
} else {
console.log(`${groupMember.userID} not muted`);
}
}
}).catch(function(imError) {
console.warn('getGroupMemberProfile error:', imError);
});
```

- For SDK versions earlier than 2.6.2, you can call the getGroupMemberProfile API to query the muting stop timestamps (muteUntil) and other profile information of group members.

Mini Program Live Streaming SDK API

Last updated : 2020-08-05 12:28:45

The live e-commerce broadcasting component results from the secondary encapsulation of Instant Messaging (IM) capabilities used in the live streaming scenario. Capabilities, such as likes, gifts, product push, and coupon claiming, are encapsulated specifically for the live streaming scenario. For how to download the related SDK, see Downloading the SDK.

Prerequisites for Using the Component

- **You have created an IM app**.
- **The following group custom field is created**.
  - attent: records the hosts that you have followed.
- **You have created the following group member custom fields**.
  - add_goods: lists newly released products on the backend in the live room.
  - room_status: controls the status of the live room.

SDK Integration

- Run the following command to integrate the IM SDK.

```
npm i tim-wx-sdk --save
```

- Run the following command to integrate the livestream marketing SDK.

```
npm i im-live-sells --save
```

Component Parameters

Parameter	Description
SDKAppID	ID of the IM app. The SDKAppID should be created in the IM console. By default, new apps are Trial Edition apps, which are mainly used for integration and testing. We recommend that you upgrade to Pro Edition or Flagship Edition before launching your app.

Parameter	Description
userSig	UserSig of the IM user. This parameter is required for logging in to the IM SDK. A UserSig can be generated at the business backend and then sent to the frontend for use. For more information on the methods and code for generating a UserSig, see Generating UserSig.
roomID	ID of the chat room. It indicates the ID of the livestreaming group (AVChatRoom) created in IM. An AVChatRoom supports an unlimited number of members and can be created in the IM Consoloe or via the REST API.
TIM	IM SDK. You need to use **tim-wx-sdk* in a Mini Program environment and ***tim-js-sdk** in a web environment.
userName	This parameter is consistent with the userName of the generated UserSig.

Initialization Example

```
import TIMLiveSell from 'im-live-sell'
import TIM from 'tim-js-sdk' // Web environment
import TIM from 'tim-wx-sdk' // Mini Program environment
const tls = new TIMLiveSell({
SDKAppID: 1400***803,
roomID: '@TGS#E****NVLGE',
userSig: 'eJwtzM9****-reWMQw_',
userName: 'Ho***st',
TIM: TIM
})
```

Component Callbacks

TLS.EVENT.SDK_READY

This callback is triggered when the initialization of the component is completed. This event corresponds to `TIM.EVENT.SDK_READY` of the IM SDK. You can call the SDK methods only after this callback is triggered.

```
tls.on(TLS.EVENT.SDK_READY, async() => {

})
```

TLS.EVENT.ROOM_STATUS_CHANGE

This callback is triggered when the room status changes, for example, when the host goes online or offline or pauses the stream.

```
tls.on(TLS.EVENT.ROOM_STATUS_CHANGE, async(data) => {

}
```

TLS.EVENT.JOIN_GROUP

This callback is triggered when a user joins the group.

```
tls.on(TLS.EVENT.JOIN_GROUP, async(data) => {
const {nick,avatar,userID} = data
})
```

TLS.EVENT.EXIT_GROUP

This callback is triggered when a user exits the group.

```
tls.on(TLS.EVENT.EXIT_GROUP, async(data) => {
const {nick,avatar,userID} = data
})
```

TLS.EVENT.NOTIFACATION

This callback is triggered when the group notification is changed.

```
tls.on(TLS.EVENT.NOTIFACATION, async(data) => {
const { notification } = data
})
```

TLS.EVENT.MESSAGE

This callback is triggered when a user sends a group message.

> Messages that you send yourself cannot be found through this callback. To display messages you sent on the screen, use the data returned by the sendMessage method. This is consistent with the IM SDK.

```
tls.on(TLS.EVENT.MESSAGE, async(data) => {
const { nick,avatar,message,userID } = data
})
```

TLS.EVENT.LIKE

This callback is triggered when a user likes the host.

```
tls.on(TLS.EVENT.LIKE, async(data) => {
const { nick,avatar,value,userID } = data
})
```

TLS.EVENT.SEND_GIFT

This callback is triggered when a user sends a gift to the host.

```
tls.on(TLS.EVENT.SEND_GIFT, async(data) => {
const { nick,avatar,value,userID } = data
})
```

TLS.EVENT.ATTENT

This callback is triggered when a user follows the host.

```
tls.on(TLS.EVENT.ATTENT, async(data) => {
const { nick,avatar,value,userID } = data
})
```

TLS.EVENT.BUY_GOODS

This callback is triggered when a user purchases a product.

```
tls.on(TLS.EVENT.BUY_GOODS, async(data) => {
const { nick,avatar,value,userID } = data
})
```

TLS.EVENT.USE_COUPON

This callback is triggered when a user claims a coupon.

```
tls.on(TLS.EVENT.USE_COUPON, async(data) => {
const { nick,avatar,value,userID } = data
})
```

TLS.EVENT.ADD_GOODS

This callback is triggered when the product recommended in the livestream changes.

```
tls.on(TLS.EVENT.ADD_GOODS, async(data) => {
const { nick,avatar,value } = data
})
```

TLS.EVENT.KICKED

This callback is triggered when the account is used for login on another device.

```
tls.on(TLS.EVENT.KICKED, async() => {

})
```

TLS.EVENT.NETWORK_CHANGE

This callback is triggered when the network changes.

```
tls.on(TLS.EVENT.NETWORK_CHANGE, async() => {

})
```

TLS.EVENT.SDK_NOT_READY

This callback is triggered when the SDK is not ready.

```
tls.on(TLS.EVENT.SDK_NOT_READY, async() => {

})
```

TLS.EVENT.PROFILE_UPDATE

This callback is triggered when the personal profile is updated.

```
tls.on(TLS.EVENT.PROFILE_UPDATE, async() => {

})
```

TLS.EVENT.ERROR

This callback is triggered when the SDK encounters an error.

```
tls.on(TLS.EVENT.ERROR, async(error) => {

})
```

${type}

This is the event with the same name that is triggered when a custom message is called.

```
tls.on(`${type}`, async(error) => {

})
```

Component Methods

sendMessage(message:string)

This method is used to send an on-screen comment.

```
/**
* When this method is called to send an on-screen comment, all users in the group can receive thi
s text message.
* @method sendMessage
* @for TLS
* @param msg - This parameter is required and indicates the on-screen comment.
* @returns Promise
*/
const {nick,avatar,message} = await tls.sendMessage(msg);
```

like(extension?:string)

This method is used to like the host.

> `extension` indicates the additional information when a like is given, for example, the user level.

```
/**
* When this method is called to give the host a like, all users in the group can receive this lik
e message.
```

```
* @method like
* @for TLS
* @param extension - This parameter is optional and indicates the additional information when a l
ike is given.
* @returns Promise
*/
const {nick,avatar} = await tls.like(extension);
```

gift(extension?:string)

This method is used to send a gift to the host.

> `extension` indicates the additional information when a gift is sent, for example, the gift information.

```
/**
* When this method is called to send a gift to the host, all users in the group can receive this
gift message.
* @method gift
* @for TLS
* @param msg - This parameter is optional and indicates the additional information when a gift is
sent.
* @returns Promise
*/
const {nick,avatar} = await tls.gift(extension);
```

exitRoom()

This method is used to exit the live room.

```
/**
* The host (group owner) cannot exit the live room.
* @method exitRoom
* @for TLS
* @param
* @returns Promise
*/
const {status} = await tls.exitRoom();
```

joinRoom()

This method is used to join a live room.

```
/**
* Join a room.
* @method joinRoom
* @for TLS
* @param
* @returns Promise
*/
const {userInfo,groupInfo} = await tls.joinRoom();
const { ownerInfo } = groupInfo;// Obtain the host information.
const { userID,nick,avatar } = ownerInfo
```

getRoomInfo()

This method is used to obtain information about the live room.

```
/**
* Obtain basic information about the live room.
* @method getRoomInfo
* @for TLS
* @param
* @returns Promise
*/
const {...ownerInfo} = await tls.getRoomInfo();
// `ownerInfo` indicates the host information.
const { userID,nick,avatar } = ownerInfo
```

attention()

This method is used to follow the host.

```
/**
* Follow the host.
* @method attention
* @for TLS
* @param
* @returns Promise
*/
const {nick,avatar} = await tls.attention();
```

cancelAttention()

This method is used to unfollow the host.

```
/**
* Unfollow the host.
* @method cancelAttention
* @for TLS
* @param
* @returns Promise
*/
const {nick,avatar} = await tls.cancelAttention();
```

destroy()

This method is used to destroy the component.

```
/**
* Destroy the component.
* @method destroy
* @for TLS
* @param
* @returns Promise
*/
tls.destroy();
```

sendCustomMsgAndEmitEvent(eventName: string, extension?: string)

This method is used to send a custom message and trigger the callback event of the specified type.

- eventName: event name
- extension: additional information about the custom message sender

```
/**
* Send a custom message and trigger an event with the same name.
* @method destroy
* @for TLS
* @param eventName: event name, someExtension: additional information
* @returns Promise
*/
await tls.sendCustomMsgAndEmitEvent('eventName','someExtension')
```

Built-in Objects

Built-in objects are created by TIM using `TIM.create` . They can use all TIM methods.

Events

Event	Description
TLS.EVENT.SDK_READY	This event is triggered when the initialization of the component is completed. This event corresponds to `TIM.EVENT.SDK_READY` of the IM SDK. You can call the SDK methods only after this event is triggered.
TLS.EVENT.JOIN_GROUP	This event is triggered when a user joins the group.
TLS.EVENT.EXIT_GROUP	This event is triggered when a user exits the group.
TLS.EVENT.NOTIFACATION	This event is triggered when the group notification is changed.
TLS.EVENT.MESSAGE	This event is triggered when a user sends a group message.
TLS.EVENT.PROFILE_UPDATE	This event is triggered when the personal profile is updated.
TLS.EVENT.ERROR	This event is triggered when the SDK encounters an error.
TLS.EVENT.KICKED	This event is triggered when the account is used for login on another device.
TLS.EVENT.NETWORK_CHANGE	This event is triggered when the network is changed.
TLS.EVENT.SDK_NOT_READY	This event is triggered when the SDK is not ready.
TLS.EVENT.LIKE	This event is triggered when a user likes the host.
TLS.EVENT.BUY_GOODS	This event is triggered when a user purchases a product.
TLS.EVENT.SEND_GIFT	This event is triggered when a user sends a gift to the host.
TLS.EVENT.ATTENT	This event is triggered when a user follows the host.
TLS.EVENT.ADD_GOODS	This event is triggered when the recommended products in a livestream change.
TLS.EVENT.USE_COUPON	This event is triggered when a user claims a coupon.