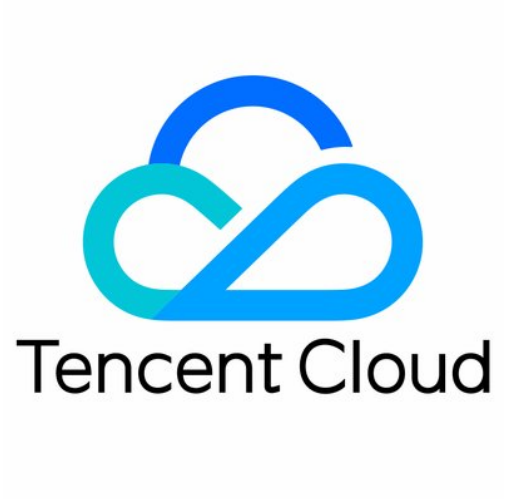
Chat

More Practices

Product Documentation



Contents

More Practices
- Live Room Setup
- AI Chatbot
- End-to-end encrypted chat with Virgil
- Super Large Entertainment and Collaboration Community
- Discord Implementation Guide
- How to Integrate Chat into Games
- Similar to the construction scheme of WeChat public account

More Practices
Live Room Setup

Last updated：2025-06-10 10:56:36

As live streaming has become ubiquitous, more and more enterprises and developers are building their own live streaming platforms. During the process, they need to handle complex requirements such as live stream push and pull, live transcoding, live screen capturing, live stream mix, live chat room, live room interaction (like, gift, and co-anchoring), and live room status management. This document takes Tencent Cloud products (Tencent Chat and CSS) as examples to describe how to implement a live room, along with possible problems and considerations, giving a glimpse of the live streaming business and requirements.
You can also quickly experience online through our Demo:

1. Preparations

Creating an Application

To set up a live room in Tencent Cloud, you need to create an Chat application in the console as shown below:

Adding a live push domain and a playback domain

The live streaming feature is required to set up a live room and can be implemented through CSS. You need to add a push domain and a playback domain as shown below:

For detailed directions, see Adding Your Own Domain.

Completing basic configuration

The application created during the preparations is the Free trial, which applies only to development. In the production environment, you need to activate the Pro edition or Pro plus edition or Enterprise edition as needed. For more information on differences between different editions, see Pricing.
In live streaming scenarios, you need some extra configurations after creating the application.

Calculating the UserSig with a key

In the Chat account system, the password required by a user login is calculated by the server with a key provided by Chat. For more information, see Generating UserSig. In the development phase, to avoid holding back development on the client, you can also calculate the `UserSig` in the console as shown below:

Configuring an admin account

During live streaming, an admin may need to send messages to a live room and mute (remove) non-compliant users, which can be done through Chat server APIs. To call these APIs, you need to create an Chat admin account. By default, Chat provides an account with the `UserID` of `administrator`. You can also create multiple admin accounts as needed. Note that you can create up to five admin accounts.

Configuring the callback address and enabling the callback

To implement lucky draws based on on-screen comments, message statistics collection, sensitive content detection, and other requirements, you need to use the Chat callback module, where the Chat backend calls back the business backend in certain scenarios. You only need to provide an HTTP API and configure it in the **Callback configuration** module in the console as shown below:

Integrating the client SDK

After completing the preparations, you need to integrate the Chat and CSS client SDKs into your project. You can select an integration option as needed.
For Chat integration, see Instant Messaging.
For CSS integration, see User Guide.
The following describes common features in a live room and provides best practices with implementation code.

2. Live Room Feature Development Guide

Live room status

A live room has the following statuses:

No.	Live Room Status
1	To be started
2	On live
3	Paused
4	Ended
5	Played back

Live room statuses have the following characteristics:

No.	Characteristics

1	Users in a live room need to be notified in real time of the change of the live room status.
2	Users new to a live room need to get the current status of the live room.

Two implementation schemes are provided, and their pros and cons are as analyzed below.

Implementation Scheme	Pros	Cons
The business backend maintains the live room status and uses the Chat server API to send custom group messages to notify users of the status.	When the live room status needs to be obtained frequently, storing the status on the business backend reduces the frequency to call the Chat SDK, compared to storing the status in the Chat group profile.This implementation scheme makes it possible to get the live room status when the Chat SDK is not integrated.	The business backend needs to provide an extra module for reading and writing the live room status.Exceptions are more likely to occur when the live room data is obtained, as the data comes from both the business backend and Chat group profile.Custom messages may be lost when sent. When there are a large number of messages, low-priority ones will be discarded first, which affects the display of the live room status. Therefore, we recommend you use high-priority custom messages.
You can store the live room status through a custom group field or group attribute and notify group users through the callback for group attribute change of the client SDK.	You don't need to provide an extra module for reading and writing the live room status.Theoretically, the callback for group attribute change won't be lost.Live room data is obtained from the group profile, which serves as a unified data source that reduces exceptions.	You need to get the group profile frequently in the high-exposure module, which increases the pressure on Chat.If the Chat SDK module is not integrated, you need to call the Chat server SDK on the business backend to get the group profile, and the number of calls is limited.

Based on the analysis above, we recommend you combine the two schemes to maintain the live room status:

1. In the live room, use scheme 1 to get the live room status.

2. Outside the live room, use scheme 2 to get the live room status.

3. After the live room status changes on the business backend, use the Chat server API to promptly sync the data to the Chat group profile (group attribute and custom group field).

Group type selection

The user chat section in live streaming scenarios has the following characteristics:

1. Users join and leave a group frequently, and group conversation information (unread count and `lastMessage`) doesn't need to be managed.
2. Users can join a group without approval.
3. Users send messages without caring about the chat history.
4. There are usually a large number of group members.
5. Group member information doesn't need to be stored.

Therefore, you can select **audio-video groups (AVChatRoom)** as the **Group type** for the live room based on the group characteristics of Chat.

IM audio-video groups (AVChatRooms) have the following features:

Support interactive live streaming scenarios with unlimited group members.

Messages (group system notifications) can be pushed to all online members.

Users can enter the group directly with no admin approval required.

Note：

The IM SDK for web allows users to join only one audio-video group (AVChatRoom) at a time. If a user logs in to an client and enters live room A, multi-client login is enabled in the console, and the user logs in to another client and enters live room B, the user will be removed from live room A.

Live room notice

The live room notice (topic) is necessary for each live room and can be seen by users after entering the room. In addition, audio-video group members need to be notified of notice changes in real time. You can store the live room notice on the business backend or in the IM group profile, just like the live room status; however, there are some additional things that require your attention:

1. The **Group Introduction** and **Group notice** fields of the **Group Profile** can contain up to **240** and **300** bytes, respectively.
2. The **Group Introduction** and **Group notice** fields of the **Group Profile` support only the string type. To create a notice with images and text, you can use the JSON string type, and the images must be accessible online URLs.
3. If a notice is set through the editor on the web, more specifically, it contains the HTML tag, it may be not parsed on the Native.

The live room notice can be set through the server APIs or the group attribute in the client SDK. The client gets the current group information through the group attribute and listens for `OnGroupInfoChange` in `GroupListener` to get the changed group attribute.

Sample code:

Android

iOS and macOS

Flutter

Web

```
// Get the group profile from the client
```

```
V2TIMManager.getGroupManager().getGroupsInfo(groupIDList, new V2TIMValueCallback<Li
  @Override
  public void onSuccess(List<V2TIMGroupInfoResult> v2TIMGroupInfoResults) {
      // Obtained the group profile successfully
  }

  @Override
  public void onError(int code, String desc) {
      // Failed to obtain the group profile
  }
});
// Modify the group profile on the client
V2TIMGroupInfo v2TIMGroupInfo = new V2TIMGroupInfo();
v2TIMGroupInfo.setGroupID("Group ID of the group to be modified");
v2TIMGroupInfo.setFaceUrl("http://xxxx");
V2TIMManager.getGroupManager().setGroupInfo(v2TIMGroupInfo, new V2TIMCallback() {
  @Override
  public void onSuccess() {
      // Modified the group profile successfully
  }

  @Override
  public void onError(int code, String desc) {
      // Failed to modify the group profile
  }
});
```

```
[[V2TIMManager sharedInstance] getGroupsInfo:@[@"groupA"] succ:^(NSArray<V2TIMGroup
    // Obtained the group profile successfully
} fail:^(int code, NSString *desc) {
    // Failed to obtain the group profile
}];
V2TIMGroupInfo *info = [[V2TIMGroupInfo alloc] init];
info.groupID = @"Group ID of the group to be modified";
info.faceURL = @"http://xxxx";
[[V2TIMManager sharedInstance] setGroupInfo:info succ:^{
    // Modified the group profile successfully
} fail:^(int code, NSString *desc) {
    // Failed to modify the group profile
}];
```

```
// Get the group profile
V2TimValueCallback<List<V2TimGroupInfoResult>> groupinfos = await groupManager.getG
// Modify the group profile
groupManager.setGroupInfo(info: V2TimGroupInfo.fromJson({
    "groupAddOpt":GroupAddOptTypeEnum.V2TIM_GROUP_ADD_AUTH
```

```
    // ...Other profiles
  }));
// Callback
TencentImSDKPlugin.v2TIMManager.addGroupListener(listener: V2TimGroupListener(onGro
    // The group information was changed.
  })));
```

```
// Get the group profile
let promise = tim.getGroupProfile({ groupID: 'group1', groupCustomFieldFilter: ['ke
promise.then(function(imResponse) {
  console.log(imResponse.data.group);
}).catch(function(imError){
  console.warn('getGroupProfile error:', imError); // Error information
});
// Modify the group profile
let promise = tim.updateGroupProfile({
  groupID: 'group1',
  name: 'new name', // Modify the group name
  introduction: 'this is introduction.', // Modify the group introduction
  // Starting from v2.6.0, group members can receive group messages about group cus
  groupCustomField: [{ key: 'group_level', value: 'high'}] // Modify the group cust
});
promise.then(function(imResponse) {
  console.log(imResponse.data.group) // Detailed group profile after modification
}).catch(function(imError){
  console.warn('updateGroupProfile error:', imError); // Error information
});
// Starting from v2.6.2, the SDK supports muting and unmuting all. Currently, when
let promise = tim.updateGroupProfile({
  groupID: 'group1',
  muteAllMembers: true, // `true`: mute all; `false`: unmute all
});
promise.then(function(imResponse) {
  console.log(imResponse.data.group) // Detailed group profile after modification
}).catch(function(imError){
  console.warn('updateGroupProfile error:', imError); // Error information
});
```

See the sample code for group profile processing in other SDKs here.

Message priority

A live room features a large number of user messages. Each user may frequently send messages. When the number of messages sent reaches the IM frequency control threshold, the IM backend will discard some messages to ensure the system running stability. It's necessary to apply such a threshold, as messages are less readable when the client receives too many of them.

Group messages are divided into three levels by priority. The backend delivers high-priority messages first. Therefore, select appropriate priorities according to the importance of messages.
The following lists the priorities from high to low:

Priority	Description
High	High priority
Normal	Medium priority
Low	Low priority

Messages are discarded according to the following policy:
Number-based frequency control limits the maximum number of messages sent per second in a single group. The default value is 40 messages per second. When the number of sent messages exceeds the limit, the backend will first deliver higher-priority messages, with messages with the same priority delivered randomly.
A message that has been restricted by frequency control is not delivered or stored in the message history, but a success response will be returned to the sender. Callback before delivering group message is triggered, but callback after delivering group message is not triggered.
Therefore, it's necessary to set message priorities for a live room.
Live room messages are prioritized based on their importance as follows:
1. Reward and gift messages that are supposed to be seen by all users.
2. Text, audio, and image messages.
3. Like messages.
Note：
In general, messages sent by client users are displayed on the screen regardless of their priority. When there are a large number of messages, it's acceptable for users to lose a small number of messages from other users.
Sample code for setting message priorities:
Android
iOS and macOS
Flutter
Web

```
// Create a custom message
V2TIMMessage v2TIMMessage = V2TIMManager.getMessageManager().createCustomMessage("C
// Set the message priority to high
v2TIMMessage.setPriority(V2TIMMessage.V2TIM_PRIORITY_HIGH)
// Send the message
V2TIMManager.getMessageManager().sendMessage(v2TIMMessage, "receiver_userID", null,
    @Override
    public void onProgress(int progress) {
        // The progress is not called back for the custom message.
```

```
    }

    @Override
    public void onSuccess(V2TIMMessage message) {
        // The custom group message sent successfully
    }

    @Override
    public void onError(int code, String desc) {
        // Failed to send the custom group message
    }
});
```

```
// Create a text message
V2TIMMessage *message = [[V2TIMManager sharedInstance] createTextMessage:@"content"
// Send the message
[V2TIMManager.sharedInstance sendMessage:message
                                receiver:@"userID"
                                 groupID:nil
                                priority:V2TIM_PRIORITY_NORMAL
                          onlineUserOnly:NO
                         offlinePushInfo:nil
                                progress:nil
                                succ:^{
    // The text message sent successfully
}
                                fail:^(int code, NSString *desc) {
    // Failed to send the text message
}];
```

```
// Create a text message
V2TimValueCallback<V2TimMsgCreateInfoResult> createTextAtMessageRes = await Tencent
    text: "test",
    atUserList: [],
  );
 if(createTextAtMessageRes.code == 0){
    String id =  createTextAtMessageRes.data.id;

       // Send the text message
    V2TimValueCallback<V2TimMessage> sendMessageRes = await TencentImSDKPlugin.v2TI
    if(sendMessageRes.code == 0){
      // The message sent successfully
    }
  }
```

```
// Send the text message
// 1. Create a message instance. The returned instance can be displayed on the scre
let message = tim.createTextMessage({
  to: 'user1',
  conversationType: TIM.TYPES.CONV_C2C,
  // Message priority, which applies to group chats and is supported by v2.4.2 or l
  // Enumerated values supported: TIM.TYPES.MSG_PRIORITY_HIGH, TIM.TYPES.MSG_PRIORI
  // priority: TIM.TYPES.MSG_PRIORITY_NORMAL,
  payload: {
    text: 'Hello world!'
  },
  // The read receipt feature is supported for one-to-one messages by v2.20.0 or la
  needReadReceipt: true
  // Message custom data (saved in the cloud, will be sent to the peer end, and can
  // cloudCustomData: 'your cloud custom data'
});
// 2. Send the message.
let promise = tim.sendMessage(message);
promise.then(function(imResponse) {
  // The message sent successfully
  console.log(imResponse);
}).catch(function(imError){
  // The message failed to be sent
  console.warn('sendMessage error:', imError);
});
```

See the sample code for message priority setting in other SDKs here.

Best practices for gift and like messages

Gift message

1. Non-persistent connection requests from the client are sent to the business server, which involves the billing logic.
2. After fees are incurred, the sender can see that XXX sent the XXX gift (so that the sender can see the gift sent by himself/herself; when there are a large number of messages, the policy for discarding messages may be triggered).
3. After the fees are settled, you can call the server API to send the custom message (gift).
4. If multiple gifts are sent in a row, you need to merge the messages.

4.1 If the number of gifts is selected in advance, for example, 99, you can send a message with `99` included in the parameter.

4.2 If gifts are sent multiple times and the total number is uncertain, you can send a message for every 20 gifts (the value can be adjusted) or the clicks within a second. For example, if 99 gifts are clicked in a row, only five messages need to be sent after the optimization.

Like message

1. Unlike a gift message, a like message is not billed and directly sent on the client.
2. For like messages that need to be counted on the server, after traffic throttling is performed on the client, likes on the client are counted, and like messages within a short period of time are merged into one. The business server gets the like count in the callback before sending a message.
3. For like messages that don't need to be counted, the logic in step 2 is used, where the business server sends a message after traffic throttling is performed on the client and doesn't need to get the count in the callback before sending a message.

Identity and level of a live room user

The concept of user level applies to most of the live rooms. You can determine a weighted user level based on the following:

1. Online duration of the live room user
2. The number of ordinary messages successfully sent by the live room user
3. The number of likes and gifts of sent by the live room user
4. Whether the user has membership privilege of the live room
5. ...

For online duration and the number of messages, you need to use the IM callbacks, which can be configured as instructed in the preparations. Callbacks in the console are as shown below:

The flowchart for information collection is as shown below:

Sample data of the callback after sending a message:

```
{
    "CallbackCommand": "Group.CallbackAfterSendMsg", // Callback command
    "GroupId": "@TGS#2J4SZEAEL", // Group ID
    "Type": "Public", // Group type
    "From_Account": "jared", // Sender
    "Operator_Account":"admin", // Request initiator
    "Random": 123456, // Random number
    "MsgSeq": 123, // Sequence number of the message
    "MsgTime": 1490686222, // Time of the message
    "OnlineOnlyFlag": 1, // The value is `1` if it is an online message and `0` (de
    "MsgBody": [ // Message body. For more information, see the `TIMMessage` messag
        {
            "MsgType": "TIMTextElem", // Text
            "MsgContent":{
                "Text": "red packet"
            }
        }
    ],
    "CloudCustomData": "your cloud custom data"
```

```
}
```

You can identify ordinary, like, and gift messages based on the message type in `MsgBody`. For more information on all the fields, see Callback After Sending a Group Message.

Sample data of the callback for the change of the user online status:

```
{
    "CallbackCommand": "State.StateChange",
    "EventTime": 1629883332497,
    "Info": {
        "Action": "Login",
        "To_Account": "testuser316",
        "Reason": "Register"
    },
    "KickedDevice": [
        {
            "Platform": "Windows"
        },
        {
            "Platform": "Android"
        }
    ]
}
```

You can identify the user online status based on the field in `Info`, so as to count the online duration. For more information on all the fields, see State Change Callbacks.

Note：

In addition, live room popularity is often displayed and used as a criterion for live room recommendation. It is determined in a similar way to user level and can be implemented through the callback system of Chat.

Historical messages in a live room

Historical messages are not stored by default for audio-video groups (AVChatRoom). When a new user enters the live room, the user can only see messages sent after the entry. To optimize the user experience, you can configure the number of historical messages that can be pulled as shown below:

Note：

This feature is available only for the Pro edition、Pro Plus edition or Enterprise edition, and up to 50 historical messages in the past 24 hours can be pulled.

Historical messages in audio-video groups are pulled in the same way as others. Sample code:

Android

iOS and macOS

Flutter

Web

```
V2TIMMessageListGetOption option = new V2TIMMessageListGetOption();
option.setGetType(V2TIMMessageListGetOption.V2TIM_GET_CLOUD_OLDER_MSG); // Pull old
option.setGetTimeBegin(1640966400);         // Start from 2022-01-01 00:00:00
option.setGetTimePeriod(1 * 24 * 60 * 60);  // Pull the messages of the whole day
option.setCount(Integer.MAX_VALUE);         // Return all the messages within the t
option.setGroupID(#you group id#);          // Pull group messages
V2TIMManager.getMessageManager().getHistoryMessageList(option, new V2TIMValueCallba
    @Override
    public void onSuccess(List<V2TIMMessage> v2TIMMessages) {
        Log.i("imsdk", "success");
    }

    @Override
    public void onError(int code, String desc) {
        Log.i("imsdk", "failure, code:" + code + ", desc:" + desc);
    }
});
```

```
// Pull historical one-to-one messages
// Set `lastMsg` to `nil` for the first pull
// `lastMsg` can be the last message in the returned message list for the second pu
[V2TIMManager.sharedInstance getC2CHistoryMessageList:#your user id# count:20 lastM
    // Record the `lastMsg` for the next pull
    V2TIMMessage *lastMsg = msgs.lastObject;
    NSLog(@"success, %@", msgs);
} fail:^(int code, NSString *desc) {
    NSLog(@"fail, %d, %@", code, desc);
}];
```

```
// Pull historical one-to-one messages
// Set `lastMsgID` to `null` for the first pull
// `lastMsgID` can be the ID of the last message in the returned message list for t
TencentImSDKPlugin.v2TIMManager.getMessageManager().getC2CHistoryMessageList(
        userID: "userId",
        count: 10,
        lastMsgID: null,
);
```

```
/ Pull the message list for the first time when a conversation is opened.
let promise = tim.getMessageList({conversationID: 'C2Ctest', count: 15});
promise.then(function(imResponse) {
  const messageList = imResponse.data.messageList; // Message list
  const nextReqMessageID = imResponse.data.nextReqMessageID; // This parameter must
```

```
  const isCompleted = imResponse.data.isCompleted; // It indicates whether all mess
});
// Pull down to see more messages.
let promise = tim.getMessageList({conversationID: 'C2Ctest', nextReqMessageID, coun
promise.then(function(imResponse) {
  const messageList = imResponse.data.messageList; // Message list
  const nextReqMessageID = imResponse.data.nextReqMessageID; // This parameter must
  const isCompleted = imResponse.data.isCompleted; // It indicates whether all mess
});
```

See the sample code for historical message pulling in other SDKs here.

Number of online users in a live room

Displaying the number of online users in a live room is a common feature. There are two implementation schemes, each with pros and cons.

1. Pull the number of online users in a group in a scheduled polling way through the getGroupOnlineMemberCount API provided by the client SDK.

2. Deliver the data to all group members through the server API, such as group system notification or custom group message, based on the callback for joining or leaving a group on the backend.

If there is only one live room, it's sufficient to pull the number of online users through the API of the client SDK. If there are multiple live rooms that require a large number of exposure positions to display the number of online users, the second scheme is recommended.

Note：

Your server can send the message of online user count statistics to the client in a scheduled manner, for example, once every five seconds. However, this will incur extra network overheads if the number of online users in the live room doesn't change sharply. We recommend you update the number by monitoring its change rate.

You can determine the priority of the accuracy and real-timeness of the number of online users in the live room as needed.

The code for getting the number of online users in a live room is as follows:

Android

iOS and macOS

Flutter

Web

```
2TIMManager.getGroupManager().getGroupOnlineMemberCount("group_avchatroom", new V2T
  @Override
  public void onSuccess(Integer integer) {
      // Obtained the number of online members in the audio-video group (AVChatRoom
  }

  @Override
  public void onError(int code, String desc) {
```

```
        // Failed to obtain the number of online members in the audio-video group (AV
    }
});
```

```
[[V2TIMManager sharedInstance] getGroupOnlineMemberCount:@"group_avchatroom" succ:^
    // Obtained the number of online members in the audio-video group (AVChatRoom)
} fail:^(int code, NSString *desc) {
    // Failed to obtain the number of online members in the audio-video group (AVCh
}];
```

```
groupManager.getGroupOnlineMemberCount(groupID: '');
```

```
// Get the number of online users in an audio-video group (supported from v2.8.0)
let promise = tim.getGroupOnlineMemberCount('group1');
promise.then(function(imResponse) {
  console.log(imResponse.data.memberCount);
}).catch(function(imError){
  console.warn('getGroupOnlineMemberCount error:', imError); // Error information
});
```

Muting a user in a live room

In a live room, you can mute all or a specified user in different scenarios.

In general, the server SDK is used for muting. To mute all, set the `MuteAllMember` field of the group attribute as instructed in Modifying the Profile of a Group. To mute a specified user, set the group member attribute as instructed in Bulk Muting and Unmuting.

When the live room admin sets group muting on the backend, the client should put the input box of the target user in the disabled status after receiving the callback event; otherwise, the user will receive the message sending failure prompt when sending a message. After the user is unmuted, the client should also put the input box in the enabled status.

Callback code on the client:

Android

iOS and macOS

Flutter

Web

```
// Mute the `userB` group member for one minute
V2TIMManager.getGroupManager().muteGroupMember("groupA", "userB", 60, new V2TIMCall
  @Override
  public void onSuccess() {
      // Muted the group member successfully
  }
```

```
  @Override
  public void onError(int code, String desc) {
      // Failed to mute the group member
  }
});

// Mute all members
V2TIMGroupInfo info = new V2TIMGroupInfo();
info.setGroupID("groupA");
info.setAllMuted(true);
V2TIMManager.getGroupManager().setGroupInfo(info, new V2TIMCallback() {
  @Override
  public void onSuccess() {
      // Muted all successfully
  }

  @Override
  public void onError(int code, String desc) {
      // Failed to mute all
  }
});

V2TIMManager.getInstance().addGroupListener(new V2TIMGroupListener() {
  @Override
  public void onMemberInfoChanged(String groupID, List<V2TIMGroupMemberChangeInfo>
    // Listen for the muting of a group member
    for (V2TIMGroupMemberChangeInfo memberChangeInfo : v2TIMGroupMemberChangeInfoLi
      // ID of the muted user
      String userID = memberChangeInfo.getUserID();
      // Muting duration
      long muteTime = memberChangeInfo.getMuteTime();
    }
  }

  @Override
  public void onGroupInfoChanged(String groupID, List<V2TIMGroupChangeInfo> changeI
    // Listen for the muting of all
    for (V2TIMGroupChangeInfo groupChangeInfo : changeInfos) {
      if (groupChangeInfo.getType() == V2TIMGroupChangeInfo.V2TIM_GROUP_INFO_CHANGE
        // Whether all members are muted
        boolean isMuteAll = groupChangeInfo.getBoolValue();
      }
    }
  }
});
```

```
// Mute the group member `user1` for one minute
[[V2TIMManager sharedInstance] muteGroupMember:@"groupA" member:@"user1" muteTime:6
    // Muted the group member successfully
} fail:^(int code, NSString *desc) {
    // Failed to mute the group member
}];

// Mute all members
V2TIMGroupInfo *info = [[V2TIMGroupInfo alloc] init];
info.groupID = @"groupA";
info.allMuted = YES;
[[V2TIMManager sharedInstance] muteGroupMember:@"groupA" member:@"user1" muteTime:6
    // Muted all successfully
} fail:^(int code, NSString *desc) {
    // Failed to mute all
}];

[[V2TIMManager sharedInstance] addGroupListener:self];
- (void)onMemberInfoChanged:(NSString *)groupID changeInfoList:(NSArray <V2TIMGroup
    // Listen for the muting of a group member
    for (V2TIMGroupMemberChangeInfo *memberChangeInfo in changeInfoList) {
      // ID of the muted user
      NSString *userID = memberChangeInfo.userID;
      // Muting duration
        uint32_t muteTime = memberChangeInfo.muteTime;
    }
}

- (void)onGroupInfoChanged:(NSString *)groupID changeInfoList:(NSArray <V2TIMGroupC
    // Listen for the muting of all
    for (V2TIMGroupChangeInfo groupChangeInfo in changeInfoList) {
      if (groupChangeInfo.type == V2TIM_GROUP_INFO_CHANGE_TYPE_SHUT_UP_ALL) {
        // Whether all members are muted
        BOOL isMuteAll = groupChangeInfo.boolValue;
      }
    }
}
```

```
// Mute the group member `userB` for ten minutes
groupManager.muteGroupMember(groupID: '',userID: 'userB',seconds: 10);

// Mute all members
groupManager.setGroupInfo(info: V2TimGroupInfo(isAllMuted: true,groupID: '',groupTy

TencentImSDKPlugin.v2TIMManager.addGroupListener(listener: V2TimGroupListener(onMem
    // The group member information is changed.
```

```
  },
  onGroupInfoChanged: (groupID,info){
    // Group profile modification
  }
  ));
```

```
tim.setGroupMemberMuteTime(options);
let promise = tim.setGroupMemberMuteTime({
  groupID: 'group1',
  userID: 'user1',
  muteTime: 600 // The user is muted for ten minutes. If the value is set to `0`, t
});
promise.then(function(imResponse) {
  console.log(imResponse.data.group); // New group profile
  console.log(imResponse.data.member); // New group member profile
}).catch(function(imError){
  console.warn('setGroupMemberMuteTime error:', imError); // Error information
});
// Set the period for muting a group member in the topic
let promise = tim.setGroupMemberMuteTime({
  groupID: 'topicID',
  userID: 'user1',
  muteTime: 600 // The user is muted for ten minutes. If the value is set to `0`, t
});
promise.then(function(imResponse) {
  console.log(imResponse.data.group); // New group profile
  console.log(imResponse.data.member); // New group member profile
}).catch(function(imError){
  console.warn('setGroupMemberMuteTime error:', imError); // Error information
});
// Starting from v2.6.2, the SDK supports muting and unmuting all. Currently, when
let promise = tim.updateGroupProfile({
  groupID: 'group1',
  muteAllMembers: true, // `true`: mute all; `false`: unmute all
});
promise.then(function(imResponse) {
  console.log(imResponse.data.group) // Detailed group profile after modification
}).catch(function(imError){
  console.warn('updateGroupProfile error:', imError); // Error information
});
```

See the sample code for group listening in other SDKs here.

Note that the change of the group member muting status will not be delivered to the client by default and needs to be configured in the console.

Note：

Client SDKs do not support user muting in live rooms at the moment. You can use the corresponding server APIs to mute and unmute users.

Banning a user in a live room

You can call the banning API on the server to kick users out of a live room and ban them from joining for a certain period of time.

Configuration items in the console are as shown below:

Note：

You can use the API for kicking users out of a live room to implement the banning feature in the client SDK on 6.6.x or later or the SDK for Flutter on 4.1.1 or later.

Sample code:

Android

iOS and macOS

Flutter

Web

```
List<String> userIDList = new ArrayList<>();
userIDList.add("userB");
V2TIMManager.getGroupManager().kickGroupMember("groupA", userIDList, "", new V2TIMV
  @Override
  public void onSuccess(List<V2TIMGroupMemberOperationResult> v2TIMGroupMemberOpera
      // Removed the member successfully
  }

  @Override
  public void onError(int code, String desc) {
      // Failed to remove the member
  }
});

V2TIMManager.getInstance().addGroupListener(new V2TIMGroupListener() {
  @Override
  public void onMemberKicked(String groupID, V2TIMGroupMemberInfo opUser,
  List<V2TIMGroupMemberInfo> memberList) {
      // A group member was removed.
  }
});
```

```
[[V2TIMManager sharedInstance] kickGroupMember:@"groupA" memberList:@[@"user1"] rea
    // Removed the member successfully
} fail:^(int code, NSString *desc) {
    // Failed to remove the member
```

```
}];

[[V2TIMManager sharedInstance] addGroupListener:self];
- (void)onMemberKicked:(NSString *)groupID opUser:(V2TIMGroupMemberInfo *)opUser me
    // A group member was removed.
}
```

```
groupManager.kickGroupMember(groupID: '',memberList: []);
```

```
tim.deleteGroupMember(options);
```

Filtering the sensitive content in a live room

Filtering the sensitive content in a live room is another important feature, which can be implemented as follows:

1. Bind the callback before sending a group message.
2. Identify the message type based on the callback data and deliver the message data to Tianyu or another third-party detection service.
3. If the message is an ordinary text message, wait for the detection result of Tianyu and return the data packet indicating whether to deliver the message to the Chat backend.
4. If the message is a rich media message, return the data packet for delivering the message to the Chat backend.

After Tianyu returns the async result, if the message is identified as non-compliant, deliver the message change notification through the message editing API or custom group message API. After receiving the notification, the client will block the non-compliant message.

Sample data of the callback before sending a message:

```
{
    "CallbackCommand": "Group.CallbackBeforeSendMsg", // Callback command
    "GroupId": "@TGS#2J4SZEAEL", // Group ID
    "Type": "Public", // Group type
    "From_Account": "jared", // Sender
    "Operator_Account":"admin", // Request initiator
    "Random": 123456, // Random number
    "OnlineOnlyFlag": 1, // The value is `1` if it is an online message and `0` (de
    "MsgBody": [ // Message body. For more information, see the `TIMMessage` messag
        {
            "MsgType": "TIMTextElem", // Text
            "MsgContent":{
                "Text": "red packet"
            }
        }
    ],
    "CloudCustomData": "your cloud custom data"
}
```

You can identify the message type based on the `MsgType` field in `MsgBody`. For more information on the fields, see Callback Before Sending a Group Message.
You can choose to handle a non-compliant message in a specific way, which can be implemented through the data packet in the callback returned to the Chat backend.

```
{
    "ActionStatus": "OK",
    "ErrorInfo": "",
    "ErrorCode": 0 // Different `ErrorCode` values have different meanings.
}
```

ErrorCode	Description
0	Speaking is allowed, and messages can be delivered.
1	Speaking is denied, and the client returns `10016`.
2	Silent discarding is enabled, and the client returns messages normally.

You can use them as needed.
The flowchart for detecting sensitive content is as shown below:

Group member list in a live room

You can call the `getGroupMemberList` API to get the list of online group members in a live room for display. As an audio-video group (AVChatRoom) has a large number of members, the feature of pulling the list of all the members is unavailable. The Pro edition、Pro Plus edition or Enterprise edition and non-Pro edition、Pro Plus edition or Enterprise edition differ in settings:
1. A non-Pro edition、Pro Plus edition or Enterprise edition user can call `getGroupMemberList` to pull the latest 30 group members.
2. An Pro edition、Pro Plus edition or Enterprise edition user can call `getGroupMemberList` to pull the latest 1,000 group members. This feature needs to be enabled in the Chat console. If it is not enabled, only the latest 30 group members will be pulled, just as in the non-Pro edition、Pro Plus edition or Enterprise edition.
The configuration in the console is as shown below:

If you are a non-Pro edition、Pro Plus edition or Enterprise edition user, you can maintain the list of online group members on the client through `getGroupMemberList` and the `onGroupMemberEnter` and `onGroupMemberQuit` callbacks of the group listening. However, after users leave and re-enter the live room, only the latest 30 group members can be pulled.
Android

iOS and macOS

Flutter

Web

```
// Use the `filter` parameter to specify only the profile of the group owner is to
int role = V2TIMGroupMemberFullInfo.V2TIM_GROUP_MEMBER_FILTER_OWNER;
V2TIMManager.getGroupManager().getGroupMemberList("testGroup", role, 0,
    new V2TIMValueCallback<V2TIMGroupMemberInfoResult>() {
    @Override
    public void onError(int code, String desc) {
        // Messages failed to be pulled
    }

    @Override
    public void onSuccess(V2TIMGroupMemberInfoResult v2TIMGroupMemberInfoResult) {
        // Conversations pulled successfully
    }
});
```

```
[[V2TIMManager sharedInstance] getGroupMemberList:@"groupA" filter:V2TIM_GROUP_MEMB
    // Conversations pulled successfully
} fail:^(int code, NSString *desc) {
    // Messages failed to be pulled
}];
```

```
// Use the `filter` parameter to specify only the profile of the group owner is to
groupManager.getGroupMemberList(count: 10,filter: GroupMemberFilterTypeEnum.V2TIM_G
```

```
tim.getGroupMemberList(options);
```

Lucky draw based on on-screen comments in a live room

Similar to message statistics, lucky draws in live streaming also require the callback after sending a message. Specifically, the message content is detected, and users that hit the keywords of a lucky draw are added to the pool.

On-screen comments for live video broadcasting

Audio-video groups (`AVChatRoom`) support on-screen comments, gifts, and like messages to build a friendly interaction experience.

Broadcast in a live room

The broadcast feature is similar to a system notice feature in the live room, but it differs from the latter in that it belongs to messaging. When the system admin delivers a broadcast message, all the live rooms under the `SDKAppID` will receive it.
The broadcast feature is currently available only for the Pro edition 、 Pro Plus edition or Enterprise edition and needs to be enabled in the console.
For detailed directions on how to send a broadcast message on the business backend, see Broadcast Messaging of Audio-Video Group.
Note：
If you are not an Pro edition 、 Pro Plus edition or Enterprise edition user, you can implement the feature by sending a custom group message on the server.

3. Live Audio/Video Stream

Anchor (stream push)

Currently, CSS offers the following ways to push a stream:
1.1 OBS (client)
1.2 Web Push (web)
1.3 MLVB SDK (app)
Note
Before starting stream push, an anchor needs to configure a push address. Refer to Push Configuration or Address Generator to generate a push address.

Client (stream pull)

The client can get live streams in different ways:
1.1 VLC for PCs
1.2 MLVB SDK for the application
Note：
You need to configure a playback address to pull a stream, which can be generated as instructed in Playback Configuration or through the Address Generator. For more information on SDK integration, see 2. Playback.
1.3

4. FAQs

1. What should I do to display the nickname and profile photo on the UI when the message sending status, `Message.nick` , and `Message.avatar` fields are empty during message sending?

Call the `getUserInfo` API to get the `Message.nick` and `Message.avatar` fields in the user's information and use them as the fields for message sending.

2. Why are messages lost?

Possible causes include the following:

An audio-video group has the frequency limit of 40 messages per second. You can check whether the callback before sending a message and the callback after sending a message are received for a lost message. If the former is received but the latter is not, then the message has been blocked due to frequency restriction.

You can check whether a group member quits on the Android, iOS, or PC client after quitting on the web client as instructed in Message.

If a problem occurs on the web client, check whether your SDK version is earlier than v2.7.6; if yes, upgrade it to the latest version.

If the problem persists, submit a ticket for assistance.

AI Chatbot

Last updated：2025-04-29 11:52:47

With the global popularity of ChatGPT, artificial intelligence (AI) has become the focus of developers today, and mainstream vendors in China have launched their own big model (BM) applications and products. Many vendors have combined their applications with AI to discover new opportunities. The powerful conversational communication capabilities of next-generation large language models (LLMs) are naturally compatible with all kinds of instant messaging scenarios, which brings broad imagination space for the combination of Tencent Chat and AI.

In office scenarios, users can chat with conversational AI to efficiently make work notes, write documents, collect information, and more. In customer service scenarios, AI-powered smart customer service can provide a conversational experience similar to human customer service and guide users to purchase and use products more effectively. In social scenarios, AI chatbots can provide users with 24-hour online psychological counseling and emotional companionship, increasing user engagement and more. Tencent Chat, the world's leading provider of communication cloud services, has also seen the huge potential of AI in the instant messaging scenario, and quickly released AI capability call APIs. Based on the communication base provided by Tencent Chat, developers can freely call industry-leading BM capabilities and empower themselves with rich AI capabilities to efficiently implement scenario-specific innovations.

This document describes how to integrate AI service capabilities into Tencent Chat through the webhook feature of Chat to build an AI chatbot for users to implement features such as intelligent customer service, creative assistance, and work assistant. (The procedure in this document takes the MiniMax LLM as an example. You can use the same method to integrate other ChatGPT-like services.)

Preparations

Creating a Tencent Chat account

Log in to your Tencent account, go to the Tencent Chat console, create an application.

Get the application's SDKAppID and key (Tencent Chat key), and create an admin account `administrator`.

Signing up for an account with the corresponding AI service provider

Sign up for an account with the provider of the AI service to be integrated, log in, and get the API key (`AI_SECRET_KEY`).

Note:

Please visit the AI service provider's official website to obtain the API-KEY，for example ：OpenAI、Gemini etc。

Creating a Tencent Chat chatbot account

Create a Tencent Chat chatbot account through the RESTful API. The Tencent Chat chatbot is a special user whose user ID begins with `@RBT#` .

```
curl -d '{"UserID":"@RBT#001","Nick":"MyRobot"}'
"https://adminapisgp.im.qcloud.com/v4/openim_robot_http_svc/create_robot?
sdkappid= {}&identifier=administrator&usersig=
{}&random=123456789&contenttype=json"
```

Replace `sdkappid={}` and `usersig={}` in the command above with your SDKAppID and the UserSig generated based on the Tencent Chat key. For more information, see Generating UserSig. After you run the command in Linux, the Tencent server returns the following information:

```
{"ActionStatus": "OK", "ErrorCode": 0, "ErrorInfo": ""}
```

The information above indicates that the chatbot `@RBT#001` with the nickname `MyRobot` was created successfully.

Configuring Tencent Chat webhooks

A Tencent Chat webhook is a request sent by the Tencent Chat backend to the backend server of the corresponding application before or after an event. The application backend can then perform the necessary data synchronization or intervene in the subsequent processing of the event. We will use a "robot event webhook" to listen for and react to user messages sent to the chatbot or @RBT# events in group chats. You need to locate and click "Robot Event Webhook" in the Tencent Chat console to enable the feature and save the settings.

Writing the Application Backend Service

Taking a one-to-one chat as an example, the overall working process is as follows:
1. The `user1` user sends the "hello" message to the chatbot `@RBT#001` .
2. The Tencent Chat backend sends a webhook to notify the application backend of the event.
3. The application backend receives the event notification which contains information such as the message sender `user1` , message recipient `@RBT#001` , and message content `hello` .
4. The application backend calls the AI service API (MiniMax API) and receives the response containing the reply message, such as "nice to meet you".
5. The application backend calls the Tencent Chat RESTful API (API `sendmsg` for a one-to-one chat and API `send_group_msg` for a group chat) to send the reply message to `user1` as `@RBT#001` .

Take the Go programming language as an example, the key code of the application backend is as follows.

Note:

The following code is for demonstration only and omits a lot of exception handling code. It cannot be used directly in production environments.

Distributing and processing the webhook command

We create an HTTP service which is listened to on port 80 and register a handler with the `/im` URL that handles all requests sent to `http://im`. All webhook requests sent by Tencent Chat contain a `CallbackCommand` parameter, with different values representing different webhook commands. The handler performs processing according to the `CallbackCommand` parameter set by Tencent Chat.

```
func handler(w http.ResponseWriter, r *http.Request) {
   command := r.URL.Query().Get("CallbackCommand")
   reqbody, _ := io.ReadAll(r.Body)
   var rspbody []byte
   switch command {
   case "Bot.OnC2CMessage": // Chatbot's webhook command word for a one-to-one mess
       dealC2c(context.Background(), reqbody)
       rspbody = []byte("{\\"ActionStatus\\": \\"OK\\", \\"ErrorCode\\": 0, \\"Erro
   default:
       rspbody = []byte("invalid CallbackCommand.")
   }
   w.Write(rspbody)
}

func main() {  // Register a handler to process the webhook command sent to the app
    http.HandleFunc("/im", handler)
    http.ListenAndServe(":80", nil)
}
```

Processing the one-to-one message received by the chatbot

When processing a one-to-one message, we first check that the sender is not a chatbot (generally a chatbot does not send a message to another chatbot) to prevent infinite webhook loops. We then parse the body of the message to obtain the content text of the message sent by the user to the chatbot, save the sender's UserID into the context to facilitate the subsequent action of calling the RESTful API to reply, and finally call `askAI` to request the AI service.

```
func dealC2c(ctx context.Context, reqbody []byte) error {
  root, _ := simplejson.NewJson(reqbody)
  jFromAccount := root.Get("From_Account")
  fromAccount, _ = jFromAccount.String()
  // Check the sender's ID to avoid processing requests sent by one chatbot to anot
  if strings.HasPrefix(fromAccount, "@RBT#") {
    return nil
```

```
  }
  jToAccount := root.Get("To_Account")
  toAccount, _ := jToAccount.String()
  msgBodyList, _ := root.Get("MsgBody").Array()
  for _, m := range msgBodyList {
    msgBody, _ := m.(map[string]interface{})
    msgType, _ := msgBody["MsgType"].(string)
    if msgType != "TIMTextElem" {
      continue
    }
    msgContent, _ := msgBody["MsgContent"].(map[string]interface{})
    text, _ := msgContent["Text"].(string)
    ctx = context.WithValue(ctx, "from", fromAccount)
    ctx = context.WithValue(ctx, "to", toAccount)
    go askAI(ctx, text)
  }
  return nil
}
```

Calling the AI service API

In this step, we use a third-party AI service, MiniMax LLM, to implement intelligent chat. Any other AI service can be used instead of the MiniMax LLM service. Note that here we demonstrate a simple `completion` API that does not contain the context of the conversation, and you can see the MiniMax documentation for details on other APIs as needed.

```
type MiniMaxRsp struct {
  Reply string `json:"reply"`
}

// Send a request to MiniMax and get a reply
func askAI(ctx context.Context, prompt string) {
  url := "https://api.minimax.chat/v1/text/completion"
  var reqData = []byte(`{
    "model": "abab5-completion",
    "prompt": prompt
  }`)
  request, _ := http.NewRequest("POST", url, bytes.NewBuffer(reqData))
  request.Header.Set("Content-Type", "application/json; charset=UTF-8
  request.Header.Set("Authorization", API_SECRET_KEY)
  client := &http.Client{}
  response, _ := client.Do(request)
  defer response.Body.Close()
  body, _ := ioutil.ReadAll(response.Body)
  rsp := &MiniMaxRsp{}
  json.Unmarshal(body, rsp)
```

```
  reply(ctx, rsp.Reply) // Send the content replied by the AI service to the user
}
```

Returning the result replied by the AI service to the user

After receiving the reply from the AI service, we only need to call the `sendmsg` RESTful API of Tencent Chat to simulate the chatbot to reply to the user, specifying the sender of the message as `@RBT#001` and the recipient as `user1`.

```
// Send a RESTful API request
func doRestAPI(host string, sdkappid int, admin, usersig, command, body string) {
  url := fmt.Sprintf("https://%s/v4/%s?sdkappid=%d&identifier=%s&usersig=%s&random=
    host, command, sdkappid, admin, usersig, rand.Uint32())
  req, _ := http.NewRequest("POST", url, bytes.NewBufferString(body))
  req.Header.Set("Content-Type", "application/json")
  cli := &http.Client{}
  rsp, err := cli.Do(req)
  if err != nil {
    log.Printf("REST API failed. %s", err.Error())
    return
  }
  defer rsp.Body.Close()
  rsptext, _ := io.ReadAll(rsp.Body)
  log.Printf("rsp:%s", rsptext)
}

// Call the RESTful API of Tencent Chat to reply to the user
func reply(ctx context.Context, text string) {
  rsp := make(map[string]interface{})
  msgbody := []map[string]interface{}{{
    "MsgType":    "TIMTextElem",
    "MsgContent": map[string]interface{}{"Text": text},
  }}
  // For the implementation of `GenUserSig`, see the Tencent  documentation.
  usersig, _ := GenUserSig(IM_SDKAPPID, IM_KEY, "administrator", 60)
  rsp["From_Account"] = ctx.Value("to").(string) //"@RBT#001"
  rsp["To_Account"] = ctx.Value("from").(string)
  rsp["SyncOtherMachine"] = 2
  rsp["MsgLifeTime"] = 60 * 60 * 24 * 7
  rsp["MsgSeq"] = rand.Uint32()
  rsp["MsgRandom"] = rand.Uint32()
  rsp["MsgBody"] = msgbody
  rspbody, _ := json.Marshal(rsp)
  doRestAPI("console.tim.qq.com", IM_SDKAPPID, "administrator", usersig, "openim/se
}
```

Effect Demonstration

The following demonstrates the final implementation effect of the Tencent Chat chatbot demo:

With the above steps, we have implemented one-to-one chat connectivity between the Tencent Chat server side and the MiniMaxAI open platform. It is also possible to integrate AI services from another AI service provider following the above steps by simply replacing the `askAI` function with the corresponding API call from that AI service provider. For group chat chatbots, only the implementation of the `Bot.OnGroupMessage` webhook command processing needs to be supplemented.

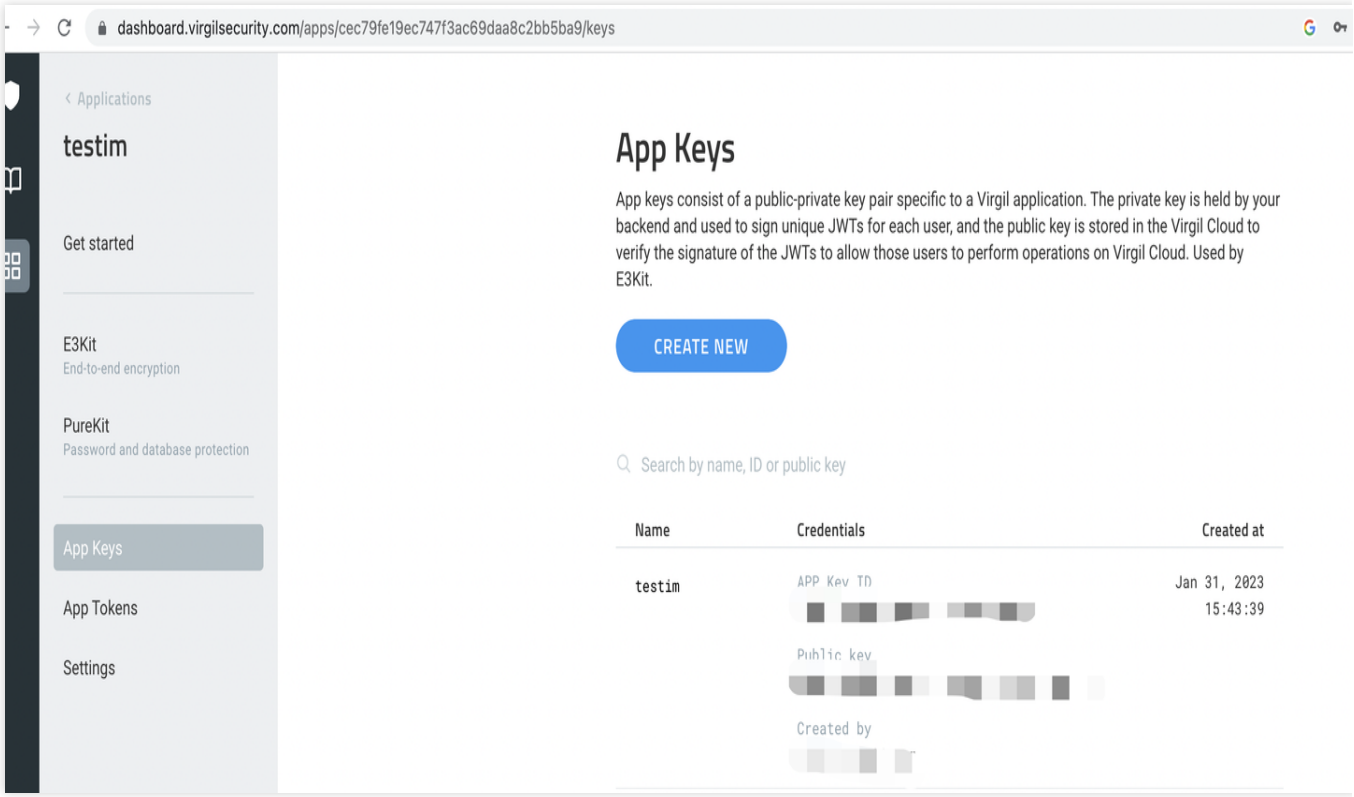
End-to-end encrypted chat with Virgil

Last updated：2024-02-07 17:30:51

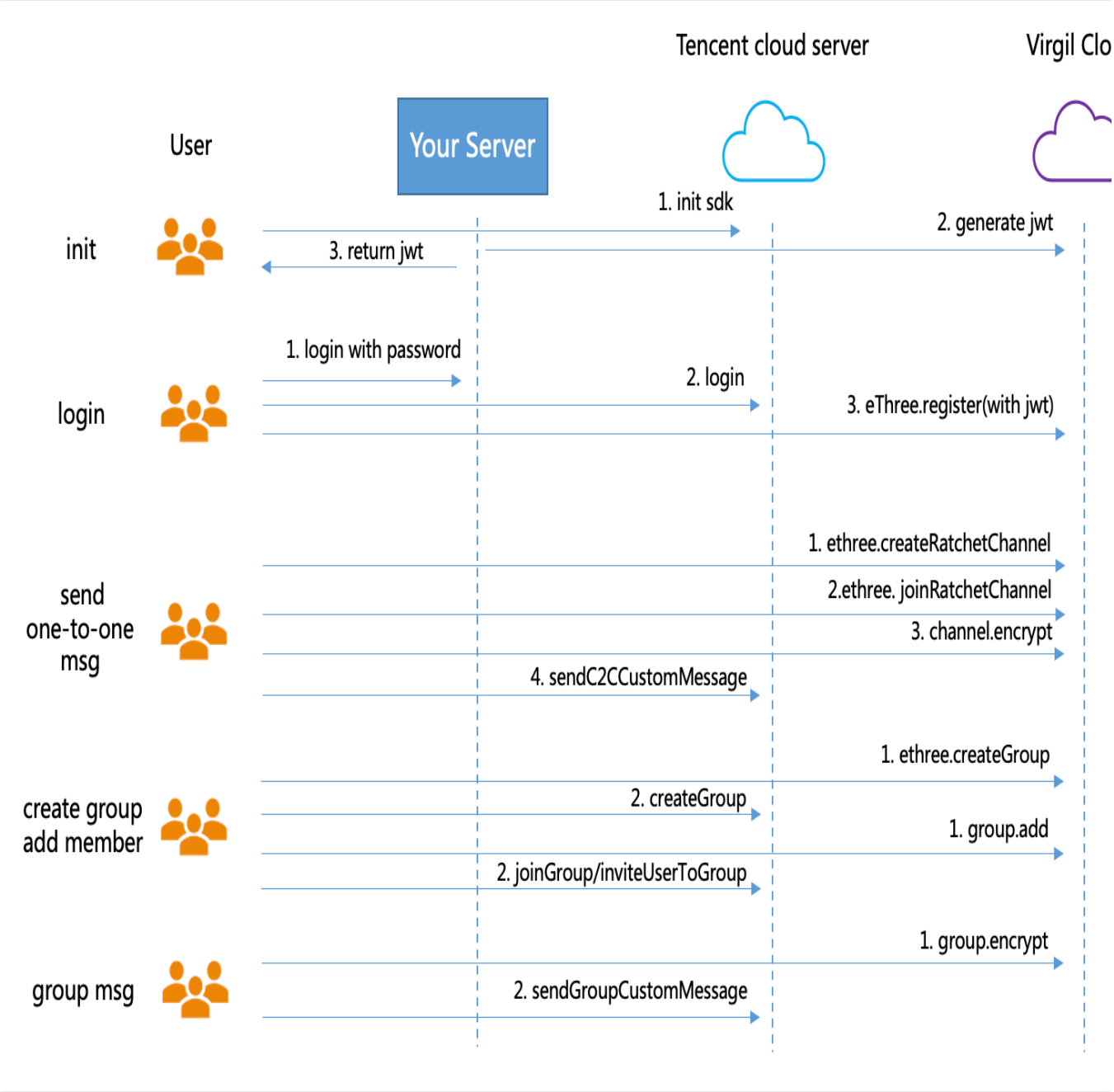
End-to-end encrypted chat with Virgil

Overview

This solution should be implemented by the customer's application based on the E3Kit of virgin security and Tencent Cloud Chat Sdk. The E3Kit document link: Virtual gild E3Kit|Virtual gild Security
Learn about the GroupEncryption-End-to-End-Encryption-E3Kit|Virtual Security documentation before reading the following solutions, especially the JWT token（JSON Web Token）, create channel/create group, encrypt and decrypt messages.
Open application in virgin security console before using. This product is a paid product. See Pricing|Virgin Security for specific pricing.



The diagram below shows the high-level communication flow：



Broad implementation overview(example:Android)

Initialization

1.Tencent Cloud imsdk initialization

```
// 1. Get the `SDKAppID` from the Chat console.
// 2. Initialize the `config` object.
V2TIMSDKConfig config = new V2TIMSDKConfig();
// 3. Specify the log output level.
config.setLogLevel(V2TIMSDKConfig.V2TIM_LOG_INFO);
```

```
// 4. Add the `V2TIMSDKListener` event listener. `sdkListener` is the
implementation class of `V2TIMSDKListener`. If you don't need to listen to IM
SDK events, skip this step.
V2TIMManager.getInstance().addIMSDKListener(sdkListener);
// 5. Initialize the IM SDK. You can call the login API as soon as you call
this API.
V2TIMManager.getInstance().initSDK(context, sdkAppID, config);
```

2.E3Kit initialization, passing in the console configuration information, and generating jwt. Corresponding operation document link: GenerateClientTokens-GetStarted-E3Kit | VirgilSecurity

The server generates jwttoken and sends it to the client

```
// generate jwt
// App Key (you got this Key at the Virgil Dashboard)
String appKeyBase64 =
"MC4CAQAwBQYDK2VwBCIEINlK4BhgsijAbNmUqU6us0ZU9MGi+HxdYCA6TdZeHjR4";
byte[] appKeyData = ConvertionUtils.base64ToBytes(appKeyBase64);

// Crypto library imports a key pair
VirgilCrypto crypto = new VirgilCrypto();
VirgilKeyPair keyPair = crypto.importPrivateKey(appKeyData);

// Initialize an access token signer that signs users JWTs
VirgilAccessTokenSigner accessTokenSigner = new VirgilAccessTokenSigner();

// Use your App Credentials you got at the Virgil Dashboard:
String appId = "be00e10e4e1f4bf58f9b4dc85d79c77a";
String appKeyId = "70b447e321f3a0fd";
TimeSpan ttl = TimeSpan.fromTime(1, TimeUnit.HOURS); // 1 hour - JWT's lifetime

// Setup a JWT generator with the required parameters:
JwtGenerator jwtGenerator =
    new JwtGenerator(appId, keyPair.getPrivateKey(), appKeyId, ttl,
accessTokenSigner);

// Generate a JWT for a user
// Remember that you must provide each user with a unique JWT.
// Each JWT contains unique user's identity (in this case - Alice).
// Identity can be any value: name, email, some id etc.
String identity = "Alice";
Jwt aliceJwt = jwtGenerator.generateToken(identity);

// As a result you get user's JWT, it looks like this:
"eyJraWQiOiI3MGI0NDdlMzIxZjNhMGZkIiwidHlwIjoiSldUIiwiYWxnIjoiVkVEUzUxMiIsImN0eS
I6InZpcmdpbC1qd3Q7dj0xIn0.eyJleHAiOjE1MTg2OTg5MTcsImlzcyI6InZpcmdpbC1iZTAwZTEwZ
TRlMWY0YmY1OGY5YjRkYzg1ZDc5Yzc3YSIsInN1YiI6ImlkZW50aXR5LUFsaWNlIiwiaWF0IjoxNTE4
```

```
NjEyNTE3fQ.MFEwDQYJYIZIAWUDBAIDBQAEQP4Yo3yjmt8WWJ5mqs3Yrqc_VzG6nBtrW2KIjP-
kxiIJL_7Wv0pqty7PDbDoGhkX8CJa6UOdyn3rBWRvMK7p7Ak".
// You can provide users with JWT at registration or authorization steps.
// Send a JWT to client-side.
String jwtString = aliceJwt.stringRepresentation();
```

The Android client initializes the E3Kit, tokenCallback is the jwttoken returned by the above server, and User1 is the user ID

```
// initialization E3Kit
// create EThreeParams with mandatory parameters
// such as identity, tokenCallback and context
EThreeParams params = new EThreeParams("User1",
   tokenCallback,context);
// initialize E3Kit with the EThreeParams
EThree ethree = new EThree(params);
```

User Login

1.Call Tencent Cloud Chat Sdk login method for account login

```
String userID = "your user id";
//Generating UserSig | Tencent Cloud
String userSig = "userSig from your server";
V2TIMManager.getInstance().login(userID, userSig, new V2TIMCallback() {
   @Override
   public void onSuccess() {
       Log.i("imsdk", "success");
   }
    @Override
   public void onError(int code, String desc) {
       // The following error codes indicate an expired `userSig`, and you need
to generate a new one for login again.
       // 1. ERR_USER_SIG_EXPIRED (6206)
       // 2. ERR_SVR_ACCOUNT_USERSIG_EXPIRED (70001)
       // Note: Do not call the login API in case of other error codes;
otherwise, the IM SDK may enter an infinite loop of login.
       Log.i("imsdk", "failure, code:" + code + ", desc:" + desc);
   }
});
```

2.Call the eThree.register method to register the user to virgilsecurity. The corresponding operation document link: UserAuthentication-E3Kit | VirgilSecurity

Note：User of im_ ID and user registered on virtilsecurity_ The id should be consistent

Start a one-to-one chat

1.Call the ethree.createRatchetChannel method in the E3Kit to create a one to one session (User1 and User2), and the corresponding operation document link: https://developer.virgilsecurity.com/docs/e3kit/end-to-end-encryption/double-ratchet/?#create -channel

User1 creates a channel with User2

```
// create one-to-one channel
ethree.createRatchetChannel(users.get("User2"))
      .addCallback(new OnResultListener<RatchetChannel>() {
          @Override public void onSuccess(RatchetChannel ratchetChannel) {
              // Channel created and saved locally!
          }

          @Override public void onError(@NotNull Throwable throwable) {
              // Error handling
          }
      });
```

User2 can join the channel

```
// join channel
ethree.joinRatchetChannel(users.get("User1"))
      .addCallback(new OnResultListener<RatchetChannel>() {
    @Override public void onSuccess(RatchetChannel ratchetChannel) {
        // Channel joined and saved locally!
    }

    @Override public void onError(@NotNull Throwable throwable) {
        // Error handling
    }
});
```

One-to-one chat message encryption and decryption

1.Use the channel created above to encrypt messages and link documents：https://developer.virgilsecurity.com/docs/e3kit/end-to-end-encryption/double-ratchet/?#encrypt-and-decrypt-messages

```
// one-to-one chat message encryption
// prepare a message
String messageToEncrypt = "Hello, User2!";

String encrypted = channel.encrypt(messageToEncrypt);
```

2.The encrypted message content is sent to Tencent Cloud Chat Sdk and sent with a customized message

```
// `msgID` returned by the API for on-demand use
String msgID = V2TIMManager.getInstance().sendC2CCustomMessage("virgil
encrypted msg", "receiver_userID", new V2TIMValueCallback<V2TIMMessage>() {
@Override
public void onSuccess(V2TIMMessage message) {
    // The one-to-one text message sent successfully
}
    @Override
public void onError(int code, String desc) {
    // Failed to send the one-to-one text message
}
});
```

3.After the peer receiving the customized message

```
// Set the event listener
V2TIMManager.getInstance().addSimpleMsgListener(simpleMsgListener);

/**
* Receive the custom one-to-one message
* @param msgID Message ID
* @param sender Sender information
* @param customData The sent content
*/
public void onRecvC2CCustomMessage(String msgID, V2TIMUserInfo sender, byte[]
customData) {
Log.i("onRecvC2CCustomMessage", "msgID:" + msgID + ", from:" +
sender.getNickName() + ", content:" + new String(customData));
//call E3Kit to decrypt msg
}
```

4.the peer calls E3Kit to decrypt and render it, as follows:

```
// Decrypt message
String decrypted = channel.decrypt(encrypted);
```

Start a channel(group)

1.Call the ethree.createGroup to create group， document link：https://developer.virgilsecurity.com/docs/e3kit/end-to-end-encryption/group-chat/?#create-group-chat

```
// create group
ethree.createGroup(groupId, users).addCallback(new OnResultListener<Group>() {
    @Override public void onSuccess(Group group) {
        // Group created and saved locally!
```

```
    }

    @Override public void onError(@NotNull Throwable throwable) {
        // Error handling
    }
});
```

2.If you need to add group members, the code below is as follows. You can call the remove method to delete a group member. Document link：https://developer.virgilsecurity.com/docs/e3kit/end-to-end-encryption/group-chat/?#add-new-participant

```
// add group member
group.add(users.get("Den")).addCallback(new OnCompleteListener() {
    @Override public void onSuccess() {
        // Den was added!
    }

    @Override public void onError(@NotNull Throwable throwable) {
        // Error handling
    }
});
```

3.Call createGroup of Tencent Cloud Chat Sdk to create a group, joinGroup or inviteUserToGroup to add group members

```
V2TIMManager.getInstance().createGroup(V2TIMManager.GROUP_TYPE_WORK, null,
"groupA", new V2TIMValueCallback<String>() {
 @Override
 public void onSuccess(String s) {
     // Group created successfully
 }
  @Override
 public void onError(int code, String desc) {
     // Failed to create the group
 }
});
// Listen for the group creation notification
V2TIMManager.getInstance().addGroupListener(new V2TIMGroupListener() {
 @Override
 public void onGroupCreated(String groupID) {
     // A group was created. `groupID` is the ID of the created group.
 }
});
```

```
// Invite the `userA` user to join the `groupA` group
List<String> userIDList = new ArrayList<>();
```

```
userIDList.add("userA");
V2TIMManager.getGroupManager().inviteUserToGroup("groupA", userIDList, new
V2TIMValueCallback<List<V2TIMGroupMemberOperationResult>>() {
 @Override
 public void onSuccess(List<V2TIMGroupMemberOperationResult>
v2TIMGroupMemberOperationResults) {
     // Invited the user to the group successfully
 }
  @Override
 public void onError(int code, String desc) {
     // Failed to invite the user to the group
 }
});
// Listen for the group invitation event
V2TIMManager.getInstance().addGroupListener(new V2TIMGroupListener() {
 @Override
 public void onMemberInvited(String groupID, V2TIMGroupMemberInfo opUser,
List<V2TIMGroupMemberInfo> memberList) {
     // A user was invited to the group. This callback can contain some UI
tips.
 }
});
```

Group chat message encryption and decryption

1.Use the group created above to encrypt messages and link documents：

https://developer.virgilsecurity.com/docs/e3kit/end-to-end-encryption/group-chat/?#encrypt-and-decrypt-messages

```
//Group message encryption
// prepare a message
String messageToEncrypt = "Hello, Bob and Carol!";

String encrypted = group.encrypt(messageToEncrypt);
```

2.The encrypted message content is sent to tencent Chat Sdk and sent with a customized message

```
String msgID = V2TIMManager.getInstance().sendGroupCustomMessage("virgil
encrypted msg ".getBytes(), "groupID", V2TIMMessage.V2TIM_PRIORITY_NORMAL, new
V2TIMValueCallback<V2TIMMessage>() {
@Override
public void onSuccess(V2TIMMessage message) {
    // The custom group message sent successfully
}
```

3.After the peer tencent cloud Chat Sdk receiving the customized message,

```
// Set the event listener
```

```
V2TIMManager.getInstance().addSimpleMsgListener(simpleMsgListener);

/**
* Receive the custom group message
* @param msgID Message ID
* @param groupID Group ID
* @param sender The group member information of the sender
* @param customData The sent content
*/
public void onRecvGroupCustomMessage(String msgID, String groupID,
V2TIMGroupMemberInfo sender, byte[] customData) {
Log.i("onRecvGroupCustomMessage", "msgID:" + msgID + ", groupID:" + groupID +
", from:" + sender.getNickName() + ", content:" + new String(customData));
//call E3Kit to decrypt msg
}
```

4.The peer decrypt and render it, as follows：

```
//Group message decryption
String decrypted = group.decrypt(encrypted, users.get("Alice"));
```

Note: After using this end-to-end encryption scheme, the local chat record search function of imsdk will not be available

IM - Developer Groups

Join a Tencent Cloud IM developer group for:

- Reliable technical support
- Product details
- Constant exchange of ideas

Telegram group (EN): join

WhatsApp group (EN): join

Telegram group (ZH): join

WhatsApp group (ZH): join

Super Large Entertainment and Collaboration Community

Last updated：2025-06-10 14:23:42

Overview

Community is a powerful tool for entertainment collaboration. It supports the **community**-**group**-**topic** hierarchy where messages are isolated and a high number of members share the same set of friend relationships. In addition, it allows you to group members and set group permissions for viewing, speaking, and managing things.

Use Cases

Finding like-minded people: a new way of user expansion

Community supports the accommodation of a high number of enthusiasts in the **community**-**group**-**topic** hierarchy. A large, open community provides more refined topics, so that users can talk and interact more freely and proactively.

Game-based social networking: higher user stickiness and engagement

Various community topics allow game players to access news, look for teammates, discuss plots, and share tips. Before the game, users can search for advice; during the game, they can always talk to others (through an always online chat room); after the game, they can dig deeper into the topics.

Fan marketing: an efficient operations tool

The diversified topics in a community can perfectly replace endless groups (such as group 1 of Shenzhen users, group 2 of Shenzhen users, group 1 of Guangzhou users, and group 1 of Shanghai users), eliminating the need to operate a number of tiring groups and making fan marketing more precise.

Organization management: a clear and layered communication method

All the members in an organization can join the same community, which supports hierarchical communication through the community hierarchy and permission settings.

Technical Strengths

A super large number of community members

The Tencent Chat community is nearly ten thousand times larger, perfectly meeting your needs to accommodate massive members in use cases such as finding like-minded people, fan marketing, game-based social networking, and organization management.

Message reliability

The Tencent Chat community greatly increases the number of members while incorporating Tencent's powerful messaging capabilities. With the complete and reliable messaging system built on over 20 years of technology accumulation, it delivers over 99.99% message sending/receiving success rate and service availability, helping you easily handle hundreds of millions of concurrent requests.

Message push performance

The Tencent Chat community adopts a new message push architecture with "fast and slow channels" plus "two-level push". This special type of architecture can well balance time and space for higher system push performance and lower terminal performance consumption. Users can enjoy the same message interaction experience in super-large groups as in general ones.

Message status and user permission management

The Tencent Chat community offers extended capabilities such as message editing, recall, and forwarding. It allows setting muting, enabling do-not-disturb, counting unread messages, and editing profiles globally or by community or topic.

Native SDK Integration Guide

Caution:

The community topic feature is supported only by the Chat native SDK Enhanced edition on v6.2.2363 or later. To use it, purchase the Pro edition 、 Pro Plus edition or Enterprise edition.

The following describes community topic APIs for Android.

Calling a community topic API

1. Call the `createGroup` API to create a topic-enabled community as instructed in the Community Management.
2. Call the `getJoinedCommunityList` API to get the list of created and joined communities.

Caution:

A community is used for managing group members but doesn't support sending or receiving messages. For more information on other community features, see the list of "other management APIs".

3. After the community is created successfully, call the `createTopicInfoCommunity` API to create topics as instructed in Topic Management.
4. Topic management also includes deleting a topic, modifying the topic information, getting the topic list, and listening for topic callbacks. In addition, topics support message sending and receiving for user communication. For more information on the APIs, see Topic Message.
5. Group community members: You can specify group information in group member custom fields to display community members in groups. To achieve this effect, all community members need to be pulled to the local for sorting by group. If the number of group members is large, it is recommended that you implement member grouping on the server side.

Web SDK Integration Guide

Caution:

The community topic feature is supported only by the Chat web SDK on v2.19.1 or later. To use it, purchase the Pro edition 、 Pro Plus edition or Enterprise edition, go to the **console**, select **Group feature configuration**, and enable the **Community** feature.

The features of the community topic APIs are as follows:

Downloading and configuring the demo source code

For a quick tryout of the Chat features, see Android.

The following describes how to use the community topic feature by calling the APIs:

Calling a community topic API

1. Call the `createGroup` API to create a topic-enabled community as instructed in the Community Management.
2. Call the `getJoinedCommunityList` API to get the list of created and joined communities. A community is used for managing group members but doesn't support sending or receiving messages. For more information on other community features, see the ordinary group APIs.
3. After the community is created successfully, call the `createTopicInfoCommunity` API to create topics as instructed in Topic Management.
4. Topic management also includes deleting a topic, modifying the topic information, getting the topic list, and listening for topic callbacks. In addition, topics support message sending and receiving for user communication. For more

information on the APIs, see Topic Messages.
5. Group community members: You can specify group information in group member custom fields to display community members in groups. To achieve this effect, all community members need to be pulled to the local for sorting by group. If the number of group members is large, it is recommended that you implement member grouping on the server side.

References

Group Management
Group System
SDK and Demo Source Code
SDK Manual
SDK Integration (Android)
SDK Integration (iOS)
SDK Integration (Web, Mini Program, and uni-app)

Contact Us

If you experience any issues, contact us.

Discord Implementation Guide

Last updated：2025-05-29 15:29:29

Discord Overview

Discord is a free online real-time messaging app and digital distribution platform designed for the communities. It allows users in the gaming, education, and business fields to communicate with each other through messages, pictures, videos, and audio in the software's chat channel.

Discord Concepts

Server

In Discord, there is a kind of group chat, which is different from the normal communication software groups, called servers (similar to communities), and server owners can create their own communities in the servers.

Channel

In servers, you can create chat channels, including the voice and text channels. The voice channel can be used for game broadcasting, chatting, etc. Channels can be set to integrate various permissions with identity groups, making the Discord community system more diverse.

Threads

Users can discuss specific topics in threads.

Preparations

Creating Tencent Cloud Chat apps

This tutorial is based on Tencent Cloud Chat. Therefore, you need to create an app in the Tencent Cloud Chat console. See the figure below.
After the app is created successfully, you can view the basic information of the app on the basic information page of the app in the Chat console.

Understanding related configuration and capabilities

To use Tencent Cloud Chat to implement Discord features, you need to understand the basic concepts related to Tencent Cloud Chat in advance and some proper terms that will be mentioned later in this tutorial, including but not

limited to the following:
SDKAppID: Tencent Cloud Chat assigns an SDKAppID for each app. The SDKAppID of an app can be viewed on the app details page after the app is created in the console and used when initializing the Chat SDK and calculating user login tickets. For more information, see UI SDK Initialization and Login.
Key: The key of an app can be viewed on the Tencent Cloud Chat console app details page and used to calculate SDK login tickets for users.
User account: Only users with Tencent Cloud Chat accounts can log in to Tencent Cloud Chat. When a user successfully logs in via a client SDK, the Chat backend automatically creates a Chat account for the user. In addition, you can use the server API that Chat provides to import the user to the user system of Chat.
Group: So far, Chat offers five types of groups for different scenarios, where users can send and receive messages.
Webhook configuration: You can proactively integrate the client and server APIs provided by Chat, and Chat will automatically return necessary information to your server when specific business logic is triggered. What you need to do is to simply complete related configuration in the Webhook Configuration module of the Chat console. Chat provides various webhook configuration and ensures high reliability for webhook events. You can use webhooks to implement various custom requirements.
Custom field: By default, Tencent Cloud Chat provides developers with fields that meet most of their needs. It also provides the following custom fields for each module to meet users' needs for extended fields:
Custom user fields
Custom friend fields
Custom group fields
Custom group member fields
Note:
To use custom fields, you can configure them in the console and then read/write them via SDK/API.

Integrating client and server SDKs

To implement Discord related features, you need to integrate Tencent Cloud Chat SDK. Chat provides diverse and easy-to-use SDKs and server APIs. Messages are synced across clients if you log in to the app with the same SDKAppID. Select an appropriate SDK according to your business requirement scenario and technology stack.

Discord Features

As the figure above shows, Discord features include servers, channels, and threads. Different servers are used for different content. For example, we have the Honor of Kings server and Game for Peace server. You can create different types of channel in a server, such as the text channel, voice channel, and notice channel. Users communicate with each other in channels. If they have more ideas on a certain topic, they can create a thread for further discussion. The following will introduce how to implement these key Discord features through Tencent Cloud Chat one by one.
Note:

For code demos, we use the Android client (Java SDK) as an example. For the calling of APIs of other SDK editions, see Integration Solution (No UI)**.**

Server

Creating a server

Analysis found the following characteristics with the Discord server list:

1. Servers can have a huge number of users.

2. Users don't actually communicate with each other in servers. Instead, they chat only in channels or threads in a server.

3. Historical chat messages in servers need to be stored on roaming servers.

4. Users have free access to servers.

5. Channels can be created in servers.

Though Chat provides five types of groups, only community groups meet the characteristics of Discord servers. A Chat community group allows users to join and leave freely, supports up to 100,000 members, and stores the message history. To join the group, a user needs to search for the group ID and send an application, and the application does not need to be approved by an admin before the user can join the group.

You can use the `createGroup` API to create a server (group). Note that the group type should be Community and setSupportTopic be `true` so that channels can be created in the server. The following is sample code:

```
V2TIMGroupInfo  groupinfo = new V2TIMGroupInfo();
groupinfo.setGroupName("Test server");
groupinfo.setSupportTopic(true);
// Initial group members
List<V2TIMCreateGroupMemberInfo> memberList = new LinkedList<V2TIMCreateGroupMember
// Other settings, such as the server profile photo
V2TIMManager.getGroupManager().createGroup(groupinfo, memberList, new V2TIMValueCal
            @Override
            public void onError(int i, String s) {
                // Creation failed
            }

            @Override
            public void onSuccess(String s) {
                // The server is created successfully, and the server ID is returne
            }
});
```

After the API is called successfully, the server ID will be returned in the `onSuccess` callback, which will be used later when you create a channel in the server.

Caution:

You can also use the server creation API provided by Chat. Key parameters are as follows:

```
{
    "Type": "Community", // Group type (required)
    "Name": "TestCommunityGroup", // Group name (required)
    "SupportTopic": 1// Whether the topic option is supported. Valid values: `1`: y
}
```

```
###### Server list

There is a list of servers that the current user has joined on the far left of
Discord. For the community scenario, Tencent Cloud Chat provides a dedicated
API for querying the server list.

```java
V2TIMManager.getGroupManager().getJoinedCommunityList(new
V2TIMValueCallback<List<V2TIMGroupInfo>>() {
            @Override
            public void onSuccess(List<V2TIMGroupInfo> v2TIMGroupInfos) {
                // The server list is got successfully, and the basic
information of the server list is provided in the returned
`List<V2TIMGroupInfo>`.
            }

            @Override
            public void onError(int i, String s) {
              // Failed to get the service list.
            }
});
```

The returned V2TIMGroupInfo provides the basic server information. However, the returned server information does not include information such as the unread message count and custom status. To achieve the same effect as that on Discord, we need to use other APIs provided by Chat to implement the unread message count for the server list, which will be discussed later in the document.

Server categories

When a server is created, a default category will be created for it. After the server is created, you can create a new category. In Tencent Cloud Chat, a server is essentially a community. Therefore, we can set a custom field for a community group to implement the server type feature. To use a custom group field, perform the steps below:

1. Enable the custom field `key` in the console.
2. User the client SDK or server API to read/write the custom field.

Set the custom group field by using the server API and client SDK.

Caution:
```

The community group feature is only available in the Pro edition 、Pro Plus edition or Enterprise edition. Before setting a custom field for the community group, you need to purchase the Pro edition 、Pro Plus edition or Enterprise edition.

Display of the unread message count and custom status for the server

As mentioned previously, the API for getting the list of joined servers does not return information such as the unread message count and server status. It should be noted that we not only need to obtain this data, but also need to listen to the change of this data to update the client UI in time. Since the server is implemented by a Chat community group, and a community group does not generate conversations in Chat, we need to count the sum of unread messages in all public and private channel conversations. We can use the getUnreadCount API of V2TIMTopicInfo to get the unread message count of public channels and use the getConversation API to get the unread message count of private channels (because private channels are implemented by work groups).

```
// Public channels
List<String> conversationIDList = new LinkedList();
conversationIDList.add("GROUP_$GROUPID");
V2TIMManager.getConversationManager().getConversationList(conversationIDList, new V
            @Override
            public void onError(int i, String s) {
                // Failed to get the conversation information of the server
            }

            @Override
            public void onSuccess(V2TIMConversationList List<V2TIMConversation>) {
               // Got the conversation information of the server successfully
            }
});
// Private channels
V2TIMManager.getGroupManager().getTopicInfoList(groupID, topicIDList, new V2TIMValu
                @Override
                public void onSuccess(List<V2TIMTopicInfoResult> v2TIMTopicInfoResu

                }

                @Override
                public void onError(int i, String s) {
                }
});
```

To implement the custom status feature for servers, we can use the setConversationCustomData API to set custom server conversation data.

```
List<String> conversationIDList = new LinkedList();
String customData = "Busy"
V2TIMManager.getConversationManager().setConversationCustomData(conversationIDList,
            @Override
            public void onSuccess(List<V2TIMConversationOperationResult> v2TIMConve
                // The custom group conversation data is set successfully
            }

            @Override
            public void onError(int i, String s) {
                // Failed to set the custom group conversation data
            }
});
```

When data related to the server conversation changes, the client needs to try to update the UI display. For listening for server conversation changes, Chat provides a corresponding event listener function addConversationListener. This callback function will be triggered in the following cases:

1. Server messages are added, deleted, or modified.
2. The unread count of server messages changes.
3. The custom server information is modified.
4. The server is pinned to the top.
5. The message receiving configuration of the server is modified.
6. The server mark is modified.
7. The server group is modified.
8. ...

```
V2TIMConversationListener conversationLister = new V2TIMConversationListener() {
            @Override
            public void onSyncServerStart() {
            }

            @Override
            public void onSyncServerFinish() {
            }

            @Override
            public void onSyncServerFailed() {
            }

            @Override
            public void onNewConversation(List<V2TIMConversation> conversationList)
```

```
            }

            @Override
            public void onConversationChanged(List<V2TIMConversation> conversationL
            }
            @Override
            public void onTotalUnreadMessageCountChanged(long totalUnreadCount) {
            }

            @Override
            public void onConversationGroupCreated(String groupName, List<V2TIMConv
            }

            @Override
            public void onConversationGroupDeleted(String groupName) {
            }

            @Override
            public void onConversationGroupNameChanged(String oldName, String newNa
            }

            @Override
            public void onConversationsAddedToGroup(String groupName, List<V2TIMCon
            }

            @Override
            public void onConversationsDeletedFromGroup(String groupName, List<V2TI
            }
    }
    V2TIMManager.getConversationManager().addConversationListener(conversationLister);
```

To sum up, we can use the `getJoinedCommunityList` and `getConversationList` APIs and the `addConversationListener` callback to implement the Discord feature of displaying the server list.

Channel

Multiple channels can be created in a server. As the figure below shows, four channels are created under the server, and the four channels are placed in two categories.

Users can invite others to join a channel and modify the basic settings of the channel. Users do most of their chatting within channels, so channel capabilities are of paramount importance in Discord. Channel capabilities in Discord correspond to topic capabilities in Tencent Cloud Chat. Chat's community groups provide the capability to create topics in groups.

Default channels

When a server is created, Discord will create four default channels for the server. Tencent Cloud Chat can also implement such a feature. The process is as follows:

1. Notify the business server that the server is created successfully via the After a group is created webhook.
2. The business side determines whether the created group is a community group, so as not to affect the business related to other groups.
3. Create a topic on the server based on the server attributes.

Parameters for creating a topic on the server are as follows:

```
{
    "GroupId": "@TGS#_@TGS#cQVLVHIM62CJ", // Group ID of the topic, which is requir
    "TopicId": "@TGS#_@TGS#cQVLVHIM62CJ@TOPIC#_TestTopic", // Custom topic ID, whic
    "TopicName": "TestTopic", // Topic name, which is required
    "From_Account": "1400187352", // Member creating the topic
    "CustomString": "This is a custom string",// Custom string
    "FaceUrl": "http://this.is.face.url", // (Optional) Topic profile photo URL
    "Notification": "This is topic Notification", // (Optional) Topic notice
    "Introduction": "This is topic Introduction" // (Optional) Topic introduction
}
```

Creating a channel

On a Discord client, users can create channels under different channel categories, as shown in the figure below:

Creating a channel in Discord is equivalent to creating a topic in a community group in Chat. When creating a topic in Chat, you can set the category and basic information of the topic.

You can use the createTopicInCommunity API to implement related features.

```
String groupID = "Server ID"
V2TIMTopicInfo info = new V2TIMTopicInfo();
info.setCustomString("{'categray':'Game','type':'text'}") // Set the channel catego
// Set the basic information for `V2TIMTopicInfo`
V2TIMManager.getGroupManager().createTopicInCommunity(groupID, info, new V2TIMValue
            @Override
            public void onSuccess(String s) {
                // Channel created successfully
            }

            @Override
            public void onError(int i, String s) {
                // Failed to create the channel
            }
});
```

When creating a channel, you can call the V2TIMTopicInfo method to set the channel information and call the setCustomString API to set the channel category and type.

When creating a channel, you can specify whether it is a private channel. A private channel is different from a general channel:
1. Users do not join a private channel after they join the server.
2. Users can join a private channel only when they are invited by the server admin.
Therefore, we can use a work group to implement the private channel feature. However, the information about the private channels bound with a server needs to be stored on the business side.

Channel types

When creating a channel in Discord, you can set the channel type to voice or text. A text channel allows chatting based on text, emojis, and images, while a voice channel allows audio/video chatting. Note that a user can be in only one voice channel at a time. To join a new voice channel, the user needs to leave the currently joined voice channel.
Pay attention to the following:
1. When creating a channel, you need to set the channel type. Find the setting method in the channel creation section.
2. When a user joins a voice channel, the system needs to determine whether the user is already in another voice channel.
3. A user can join only one voice channel within an application.
4. Tencent Cloud Chat currently does not provide an API to query whether a user is in a voice channel and in which voice channel. This part of data needs to be maintained on the business side.
Regarding the last point above, developers can use the group joining and leaving callbacks provided by Tencent Cloud Chat to maintain the voice channel statuses of users and store the statuses on the business side. Note that the callbacks provided by Tencent Cloud Chat may have delays, which means that after a user leaves one voice channel, the user may not be able to join another voice channel for a short period of time. Therefore, to address this issue, we can also use the client to report the user's voice channel joining or leaving status to the business server in real time.

Inviting a user to a channel

There are three ways a user can join a channel:
1. Be invited to join a server by another user. When a user joins a server, the user joins all the server's public channels.
2. Search for a server and then join the server.
3. Be invited by the server admin to join a private channel.
According to the characteristics of the community, users only need to join a server to join public channels.
1. Supports joining a server by accurate group ID search.
2. Supports joining a server via an invitation.
3. Supports proactively applying for joining a server, and approval is not required.

Setting a channel

You can mute or unmute a channel and configure notifications for a channel. The corresponding APIs are setAllMute and setGroupReceiveMessageOpt respectively.

```
// Set the basic channel information
V2TIMManager.getGroupManager().setTopicInfo(topicInfo, new V2TIMCallback() {
    @Override
    public void onSuccess() {
        // Configured successfully
    }

    @Override
    public void onError(int i, String s) {
        // Failed to configure
    }
});
```

```
// Set the message receiving option for the channel
String groupID = "topicid"
int opt = 0;
V2TIMManager.getMessageManager().setGroupReceiveMessageOpt(groupID, opt, new V2TIMC
    @Override
    public void onSuccess() {

    }

    @Override
    public void onError(int i, String s) {

    }
});
```

Channel message list

You can use the getHistoryMessageList API to get the historical messages of a channel.

```
final V2TIMMessageListGetOption option = new V2TIMMessageListGetOption();
option.setGroupID("Channel ID");
option.setCount(20);
// Other settings
V2TIMManager.getMessageManager().getHistoryMessageList(option, new V2TIMValueCallba
    @Override
    public void onSuccess(List<V2TIMMessage> v2TIMMessages) {
        // The historical messages of the channel are got successfully
    }

    @Override
    public void onError(int code, String desc) {
        // Failed to get the historical messages of the channel
```

```
    }
});
```

Unread message count of a channel

The unread message count of a channel is different from that of a server. The unread message count of a server is in the conversation information, while the unread message count of a channel is in the channel basic information. We can use the group callback onTopicInfoChanged provided by Tencent Cloud Chat to get the unread message count in real time.

```
String groupID = "Server ID";
List< String > topicIDList= new LinkedList(); // List of channel messages
V2TIMManager.getGroupManager().getTopicInfoList(groupID, topicIDList, new V2TIMValu
    @Override
    public void onSuccess(List<V2TIMTopicInfoResult> v2TIMTopicInfoResults) {
        // Get the channel information, such as the channel ID, name, and unread me
    }

    @Override
    public void onError(int i, String s) {

    }
});
```

Channel member list

A user that joins a server will join all public channels under the server by default. Therefore, to get the public channel member list, you only need to get the group member list of a server. The following shows how to get the public channel member list:

```
String groupID = "Server ID";
int filter = 0; // Group member role: admin, ordinary members...
long nextSeq = 0;// Pagination parameter
V2TIMManager.getGroupManager().getGroupMemberList(groupID, filter, nextSeq , new V2
    @Override
    public void onError(int i, String s) {
        CommonUtil.returnError(result,i,s);
    }

    @Override
    public void onSuccess(V2TIMGroupMemberInfoResult v2TIMGroupMemberInfoResult) {

    }
});
```

To get the member list of a private channel, you can also call the getGroupMemberList API but pass in the group ID of the private channel.

ID Card Permissions

For permission groups, users can define permissions for groups as needed and configure different permissions and members for different permission groups, so that users can manage groups by permissions. Community groups can be managed by permission groups. Management of permission groups is more flexible than that of admins with fixed roles, making it suitable for communities with many members and topics.

For examples of integrating permission groups on the client side, see: No UI Integration > Community Topics > Permission Group.
For examples of integrating permission groups on the server side, see: Server-side API > REST API > Permission Group Management > Create Permission Group.

Threads

A thread is a discussion group for a specific topic in a channel. Users can browse messages on a specific topic within a thread, and the digest of the thread can also be viewed in the message list of the channel. A thread is also considered a group in Tencent Cloud Chat.

Creating a thread

A thread can be created separately or by binding with a message in a channel. You can use the `createGroup` API provided by Tencent Cloud Chat to create a thread. Pay attention to the following:
1. When a thread is created, all members of the current server are in the thread by default.
2. Members who join the server after the thread is created will also be added to the thread.
3. Users who later join the thread can view the thread's chat history since its creation.
To sum up, when a thread/group is created, the server's group members are added to the thread, and when a user joins the server, we need to add the user to all the threads of the server in the user's group joining callback. It's best to implement these two steps on the business server side, using the following APIs:
1. Querying members in a group
2. Creating a group and setting group members
3. Inviting users to join a group
The server-side webhook is the webhook After a User Joins a Group.

Number of threads

In the channel list, you can get the list of threads in the channel. Therefore, we need to set up the many-to-one relationships between threads and the server. If threads have historical messages, we also need to set up the one-to-one relationships between the threads and messages. Tencent Cloud Chat currently does not provide capabilities to set up these relationships, so these relationships need to be maintained on the business side. We can use the HTTP

API provided by the business side to get the number of threads and the thread list and send online messages to update the thread list in real time.

Thread digests

When a user creates a thread for a message, the user can view the following message digest information in the message list:
1. Number of messages in the thread
2. `lastMessage` related information in the thread
3. Other information that needs to be displayed in the message list in real time
To implement the message digest capability of threads, we need to use the group message sending callback and message editing capabilities. After a user sends a message in a thread, the Chat service triggers the message sending callback and synchronizes related information to the business side. Then the business counts the number of messages of the thread, maintains the `lastMessage` information, and stores the information to the message via the message editing API.

Messages

Message feedback

Message feedback is the extension of a message for users, as shown in the figure below:
Because all types of messages support editing and feedback and require community group support, we recommend that the data be stored in the `cloudCustomData` field. The following is the detailed data storage format for your reference:

```
{
  "reaction": {
    "simle":["user1","user2"]
  }
}
```

```
V2TIMManager.getMessageManager().modifyMessage(modifiedMessage, new V2TIMCompleteCa
    @Override
    public void onComplete(int i, String s, V2TIMMessage v2TIMMessage) {
        // Message modification completed
    }
});
```

Message editing

Message editing works the same as message feedback, with only a few differences in the custom data configured. To enable all messages to support editing, we recommend that the data be edited in the `cloudCustomData` field.
The data format is as follows:

```
{
  "isEdited": true
}
```

```
V2TIMManager.getMessageManager().modifyMessage(modifiedMessage, new V2TIMCompleteCa
    @Override
    public void onComplete(int i, String s, V2TIMMessage v2TIMMessage) {
        // Message editing completed
    }
});
```

Message starring

Message starring is to star a message in a group chat so that other users can see the starred message. Since community groups do not support group attributes, we use custom messages to implement the message starring capability.

To use custom group messages, we need to make configuration as shown in the figure below in the Tencent Cloud Chat console first:

Then make the following configuration:

```
V2TIMGroupInfo info =  new V2TIMGroupInfo();
info.setCustomInfo("pin data");
V2TIMManager.getGroupManager().setGroupInfo(info, new V2TIMCallback() {
    @Override
    public void onError(int i, String s) {
        // Failed to configure
    }

    @Override
    public void onSuccess() {
        // Configured successfully
    }
});
```

Then, group members can receive the onMemberInfoChanged event, and offline users can also get the starred message content via the group profile:

```
List< String > groupIDList = new LinkedList();
V2TIMManager.getGroupManager().getGroupsInfo(groupIDList, new V2TIMValueCallback<Li
    @Override
    public void onError(int i, String s) {

    }

    @Override
```

```
    public void onSuccess(List<V2TIMGroupInfoResult> v2TIMGroupInfoResults) {

    }
});
```

Note that custom fields currently can be configured only on the server, not on channels.

"Typing..." status

When a friend is typing, users on other clients can see the user's "typing..." status, as shown in the figure above. We can use the online message feature of Tencent Cloud Chat to implement this capability. The "typing..." status message:

1. Can be received only by online users.
2. Will not be stored on a roaming server.

In addition, we've made the following optimizations for better performance:

The sending of the "typing..." status message will only be triggered if two users send messages to each other within 20s.

The same text message does not trigger online message sending multiple times.

Private messages

Private messages are messages that Discord users send to each other, regardless of whether they are friends or not. To send a private message to another user that is not your friend, you need to know the user's userID and disable the relationship chain check for message sending option in the Tencent Cloud Chat console. Otherwise, message sending will fail.

Alternatively, you can call the addFriend API to add that user as your friend and use the getFriendList API to get your contacts.

The relevant code is as follows:

```
// Add a friend
V2TIMManager.getFriendshipManager().addFriend(info, new V2TIMValueCallback<V2TIMFri
            @Override
            public void onError(int i, String s) {
                // Failed to add the friend
            }

            @Override
            public void onSuccess(V2TIMFriendOperationResult v2TIMFriendOperationRe
               // Friend added successfully
            }
});
```

```
// Obtain the contacts
V2TIMManager.getFriendshipManager().getFriendList(new V2TIMValueCallback<List<V2TIM
```

```
            @Override
            public void onError(int i, String s) {
                // Failed to obtain the contacts
            }

            @Override
            public void onSuccess(List<V2TIMFriendInfo> v2TIMFriendInfos) {
                // The contacts obtained successfully
            }
});
```

Personal center

Online status

The online status capabilities of Discord's user panel can be implemented by the following online status APIs provided by Tencent Cloud Chat:

setSelfStatus: API for setting one's own custom online status. Users' online/offline statuses are set by Tencent Cloud Chat and cannot be modified by developers.

getUserStatus: API for getting the online statuses of friends.

onUserStatusChanged: API for listening for the online status changes of friends.

The relevant code is as follows:

```
// Set your own online status
V2TIMUserStatus customStatus = new V2TIMUserStatus();
V2TIMManager.getInstance().setSelfStatus(customStatus, new V2TIMCallback() {
    @Override
    public void onSuccess() {

    }

    @Override
    public void onError(int i, String s) {

    }
});
```

```
// Get the online statuses of friends
List<String> userIDList = new LinkerList();
V2TIMManager.getInstance().getUserStatus(userIDList, new V2TIMValueCallback<List<V2
    @Override
    public void onSuccess(List<V2TIMUserStatus> v2TIMUserStatuses) {

    }
```

```
    @Override
    public void onError(int i, String s) {

    }
});
```

Personal information

Personal information related features can be implemented by the following relationship chain related APIs provided by Tencent Cloud Chat:

getUsersInfo: API for getting user profiles

setSelfInfo: API for setting the user's profile

```
// Set the user's profile
final V2TIMUserFullInfo userFullInfo = new V2TIMUserFullInfo();
V2TIMManager.getInstance().setSelfInfo(userFullInfo, new V2TIMCallback() {
    @Override
    public void onError(int i, String s) {

    }

    @Override
    public void onSuccess() {

    }
});
```

```
// Get user profiles
List<String> userIDList = new LinkedList();
V2TIMManager.getInstance().getUsersInfo(userIDList, new V2TIMValueCallback<List<V2T
    @Override
    public void onError(int i, String s) {

    }

    @Override
    public void onSuccess(List<V2TIMUserFullInfo> v2TIMUserFullInfos) {

    }
});
```

Others

Search

Discord provides the following search capabilities:
Tencent Cloud Chat provides rich search capabilities, including:
searchLocalMessages: API for searching for messages
searchGroups: API for searching for groups
searchGroupMembers: API for searching for group members
searchFriends: API for searching for friends
For how to use the APIs, see the official documentation.

Offline push

Tencent Cloud Chat's online messages cannot reach offline users. To address this issue, we need to integrate the offline push capabilities provided by device vendors to Tencent Cloud Chat. For more information, see Offline Push.

Sensitive word verification

When you send information and configuration in Discord, the content needs to be filtered. Tencent Cloud Chat also provides a similar scheme to help users use Chat with compliance.

How to Integrate Chat into Games

Last updated：2025-05-29 15:29:00

adminapisgp.im.qcloud.comadminapisgp.im.qcloud.comInstant messaging is a common requirement of games, and instant chat has become an essential feature in multiplayer games. The game platform itself involves a variety of group types, custom message types (such as in-game props gifting and trading), global access, and other complex requirements. This document sorts out the implementation methods of common requirements in the process of building in-game chat, along with possible problems and considerations, helping developers quickly understand the business and implement their requirements.

Preparations

Calculating the UserSig with a key

In the Chat account system, the password required by a user login is calculated by the server with a key provided by Chat. For more information, see Generating UserSig. In the development phase, to avoid holding back development on the client, you can also calculate the `UserSig` in the console as shown below:

Configuring an admin account

Managing in-game instant messaging may require admins to send email announcements to games, manage temporary team-up messages, and so on, which can be done through Chat server APIs. To call these APIs, you need to create a Chat admin account. By default, Chat provides an account with the `UserID` of `administrator`. You can also create multiple admin accounts as needed. Note that you can create up to five admin accounts.

Configuring the callback address and enabling the callback

To implement in-game team-up and other requirements, you use the Webhook event, where the Chat backend calls the business backend in certain scenarios. You only need to provide an HTTP API and configure it in the **Webhook Configuration** module in the console as shown below:

Integrating the client SDK

After completing the preparations, you need to integrate the Chat client SDK into your project. You can select different integration options as needed. For detailed directions, see TUIKit Introduction.

The following describes common Chat features to be integrated into games and provides best practices with implementation code.

Game Chat Room Feature Development Guide

User profile

Common user profile

Common user profiles stored in gaming business can be divided into basic information profiles and other information profiles.

Basic Information	Other Information
Username, gender, date of birth, level, role, mobile number, etc.	Other information required by games

Profile storage

The gaming business has a large number of users and it is difficult to store huge amounts of user data. Tencent Cloud Chat offers a complete set of profile solutions through its user profile hosting capabilities. The following compares storing user profiles to Tencent Cloud Chat and to the business background.

Item	Chat	Business Background
Storage capacity	Supports automatic scaling	Supports limited capacity and difficult to scale
User profiles	Supports standard and custom fields, with restrictions on the lengths and names of the fields	Customizable, more flexible
Profile read/write	Supports easy-to-use service APIs and guidelines	Requires development on your own
APIs	Requires the API call frequency to be 200 times/second or less	Allows you to develop API call and other capabilities as needed
Security	Supports remote disaster recovery and cross-region deployment	Requires maintenance on your own

In addition to profile storage and read/write capabilities, Chat has the following advantages thanks to its user profile hosting service:
1. Chat provides remote disaster recovery, cross-region deployment, and auto scaling capabilities. In this way, you are completely free from complex processing flows, such as server downtime, multi-copy primary-secondary replication, and capacity scaling.

2. Chat provides the business processing flows commonly used in the industry, with which you do not have to worry about user profile business logic.
3. Chat provides professional operation processes and teams, ensuring 99.99% service quality annually and helping you offer services known for their stability.
4. Chat provides easy-to-use service APIs and easy-to-access guidelines, with premium services throughout the whole process.

Chat user profile storage methods

The Chat storage scheme includes user profile storage and read/write capabilities. The following describes how Chat stores user profile, friend profile, and extension data. All data is stored in `Key-Value` format. `Key` is of the `String` type and supports only uppercase and lowercase letters, digits, and underscores. `Value` supports the following types:

Type	Description
uint64_t	Integer (Not supported by custom fields.)
string	String. The string length cannot exceed 500 bytes.
bytes	Buffer. The buffer length cannot exceed 500 bytes.
string array	String array. Each string cannot exceed 500 bytes. Available only to the `Tag_SNS_IM_Group` field of the friend list.

User profile: A user profile includes standard and custom fields. For the custom fields, see the extension data part below. For the standard fields currently supported by Chat, see Standard Profile Fields.
Friend profile: A friend profile includes standard and custom friend fields. Chat's contact list supports up to 3,000 friends. The standard friend fields are described as follows:

Field Name	Type	Description
Tag_SNS_IM_Group	Array	Friend list
Tag_SNS_IM_Remark	String	Friend remarks
Tag_SNS_IM_AddSource	String	Source of the friend request
Tag_SNS_IM_AddWording	String	Content of the friend request
Tag_SNS_IM_AddTime	Integer	Timestamp of the friend request

For more information, see Standard friend fields.

Custom profile: To apply for custom profile fields, the app admin can log in to the Chat console and find your app and go to **Feature Configuration**. After the application is submitted, custom profile fields will take effect in five minutes.
For more information on user profile management, see Profile Management.
For more information on relationship chain custom profile management, see Custom friend fields.

Chat user profile storage limits

Service feature limits

Data storage: Each string or buffer cannot exceed 500 bytes.
Custom fields: The keywords of custom fields must consist of English letters with a length no longer than 8 bytes. The values of custom fields cannot exceed 500 bytes.
Friend relationship chain: Each user can have up to 3,000 friends.

API-related limits

Account management: Up to 100 usernames can be imported at a time, and the status of up to 500 users can be queried per request.
Other call frequency: Up to 200 times per second.
For more information on the use limits, see Use Limits.

Email systems

Email systems are almost mandatory in games these days. Emails can contain text messages, as well as email attachments such as game props and rewards. An email can be sent to a single user, or it can be sent as a group email to give out event rewards. The following describes how the features of an email system are implemented from diverse aspects such as players receiving emails, email list, email unread count, sending emails to all members, and email validity period.

Email receiving and sending

Players receiving emails: When system emails are sent successfully and players are online, the players can receive the system emails properly. The players can obtain the historical or latest emails by getting the message list of email conversations. They can also add or delete the listener for the receiving of new messages of all types (including text, custom, and rich media messages). The sample code for Unity is as follows:

```
// Configure the message receiving event listener
TencentIMSDK.AddRecvNewMsgCallback((List<Message> messages, string user_data)=>{
foreach(Message message in messages)
{
  foreach (Elem elem in message.message_elem_array)
  {
    // There is a next element
    if (elem.elem_type == TIMElemType.kTIMElem_Text)
    {
       string text = elem.text_elem_content;
    }
```

```
    }
}
})
// Listen for the `RecvNewMsgCallback` callback to receive messages
// To stop receiving messages, call `RemoveRecvNewMsgCallback` to remove the listen
```

For more information, see Unity - Receiving Message.

System sending emails: The system can send system emails to users via different server APIs, which are described as follows:

API	Characteristics	Application Scenarios
Sending one-to-one messages to one user	Sends a message to a specified account. The sender displayed to the recipient is not the admin, but the account specified by the admin.	Sending messages to a specific user, for example, sending a game rewarding message to a user
Sending one-to-one messages to multiple users	A one-to-one message can be sent to up to 500 users at a time, and the highest call frequency is 200 times per second.	Sending messages to specific users, without the need to create a recipient group. If the number of recipients is large, you need to send the messages in batches.
Sending ordinary messages in a group	When sending ordinary messages to a group, you need to add all recipients to the same group.	Sending ordinary messages to a large number of users (up to 100,000 users per community group)
Pushing to all users	Pushes messages to all users in an app. You can specify user tags and attributes for message sending.	Pushing messages to all users in an app or to a large number of users with specific characteristic attributes, such as campaign emails

Note:

Pushing to all users is available only to the Pro edition、Pro Plus edition or Enterprise edition. To use it, you need to purchase the Pro edition、Pro Plus edition or Enterprise edition, go to the console, choose **Feature Configuration** > **Login and Message** > **Push to all users**, and enable the feature.

The following is a sample of a basic request for sending an ordinary message in a group:

```
{
    "GroupId": "@TGS#2C5SZEAEF",
    "Random": 8912345, // A random number. If the random numbers of two messages ar
    "MsgBody": [ // Message body, which consists of an element array. For details,
        {
            "MsgType": "TIMTextElem", // Text
            "MsgContent":{
```

```
            "Text": "red packet"
            }
        },
        {
            "MsgType": "TIMCustomElem", // Custom
            "MsgContent":{
            "Data": "message",
            "Desc": "notification",
            "Ext": "url",
            "Sound":"dingdong.aiff"
            }
        }
    ],
}
```

`MsgBody` (message body) is a message array. You can add text and custom messages to be sent to it.

Email list

The storage of the historical email list is equivalent to the storage of messages. It consists of the storage of historical one-to-one messages and the storage of historical group messages. Because a group chat requires at least two users, you can create a group containing the account specified by the admin and the user receiving the email for storage.

Note:

For the Free Trial and Standard edition, the storage period is seven days. For the Pro edition, the storage period is 30 days,For the Pro Plus edition、Enterprise edition the storage period is 90 days. The Pro edition and Pro Plus edition、Enterprise edition allow you to extend the storage period. You can log in to the console to modify related configuration. Extending the storage period of historical messages is a paid value-added service. For more information on billing, see Value-added Service Pricing.

When the network is normal, the latest cloud data will be pulled; when it is abnormal, the SDK will return the locally stored historical messages. Paged pulling is supported. The following is Unity sample code for pulling the historical email list:

```
// Pull historical one-to-one messages
// Set `msg_getmsglist_param_last_msg` to `null` for the first pull
// `msg_getmsglist_param_last_msg` can be the last message in the returned message
var get_message_list_param = new MsgGetMsgListParam
{
    msg_getmsglist_param_last_msg = LastMessage
};
TIMResult res = TencentIMSDK.MsgGetMsgList(conv_id, TIMConvType.kTIMConv_C2C, get_m
// Process the callback logic
});
```

For more information on how to pull historical messages, see Historical Message - Unity.

Unread email count

A record of a user-system email is equivalent to a conversation in a chat. Tencent Cloud Chat provides a conversation unread count feature to remind users that they have not yet read messages. When a user clicks into the conversation and returns to the conversation list, the unread message count is cleared. The following is Unity sample code:

```
// Get the total unread count
TIMResult res = TencentIMSDK.ConvGetTotalUnreadMessageCount((int code, string desc,
 // Process the async logic
});

// Unread count change notification
TencentIMSDK.SetConvTotalUnreadMessageCountChangedCallback((int total_unread_count,
 // Process the callback logic
});

// Clear the unread count of all conversations
TIMResult res = TencentIMSDK.MsgMarkAllMessageAsRead((int code, string desc, string
 // Process the async logic
});
```

For more information, see Unity - Conversation Unread Count.

Sending emails to all members

Sending emails to all members is to send email messages to all players in a game. Tencent Cloud Chat provides the pushing to all members feature at the server side. The following is sample code:

```
https://adminapisgp.im.qcloud.com/v4/all_member_push/im_push?
usersig=xxx&identifier=admin&sdkappid=88888888&random=99999999&contenttype=json
```

The following is a sample request:

```
{
    "From_Account": "admin",
    "MsgRandom": 56512,
    "MsgLifeTime": 120, // Offline storage for 120s (two minutes)
    "MsgBody":[
        {
        "MsgType": "TIMTextElem",
        "MsgContent":{
            "Text": "hi, beauty"
            }
        }
    ]
}
```

The feature of sending messages to all members allows you to set the offline storage period, so that even if some users are not online, they can still receive the messages within the specified offline storage period. Configure `MsgLifeTime` (unit: second) to specify the offline storage period: The maximum period is 604,800 seconds (seven days). The default value is `0`, which indicates that messages are not stored offline.
For more information on pushing to all users, see Pushing to All Users.

Email validity period

For the Free Trial and Standard edition, the storage period is seven days. For the Pro edition, the storage period is 30 days,For the Pro Plus edition、Enterprise edition the storage period is 90 days. The Pro edition and Pro Plus edition and E enterprise edition allow you to extend the storage period. For more information on the storage of historical emails, see Historical message storage period settings.
In addition, when sending a message on the server side, you can set `MsgLifeTime` to specify the offline storage period (up to seven days) of the message. If `MsgLifeTime` is `0`, it indicates that the message is sent to online users only and is not stored offline. For more information, see here.

Temporary team-up

Temporary team-up is essential in online multiplayer games. The following describes temporary team-up scenarios and how the background and team members obtain team information.

Team-based scenarios

Team-based scenarios include creating a team, leaving a temporary team, becoming the team leader, inviting others to join a team, and disbanding a team. The following are some code examples to explain the implementation of different scenarios.
Creating a team before the game begins: When the first player enters the game, the server automatically creates a group and the maximum number of group members can be specified. If the request specifies the group owner or group members, the specified group owner or group members will be automatically added to the group when the group is created. The following is a sample request URL:

```
https://adminapisgp.im.qcloud.com/v4/group_open_http_svc/create_group?
sdkappid=88888888&identifier=admin&usersig=xxx&random=99999999&contenttype=json
```

The following is a basic request sample:

```
{
  "Owner_Account": "leckie", // UserId of the group owner (optional)
  "Type": "Public", // Group type: Private, Public, ChatRoom, AVChatRoom, or Commun
  "Name": "TestGroup", // Group name (required)
  "MaxMemberCount":5 // Maximum number of group members (optional)
}
```

Note:

An app supports up to 100,000 groups. For extra groups, you need to pay certain fees, as described in Pricing. For more information on group creation on the server side, see Creating a Group.

Adding group members: If new players enter the game after the group chat is created, you need to add the new players to the existing group. The following is a sample request URL:

```
https://adminapisgp.im.qcloud.com/v4/group_open_http_svc/add_group_member?
sdkappid=88888888&identifier=admin&usersig=xxx&random=99999999&contenttype=json
```

The following is a request sample:

```
{
  "GroupId": "@TGS#2J4SZEAEL", // The group to which to add members (required)
  "MemberList": [ // Up to 300 members can be added at a time.
      {
          "Member_Account": "tommy" // The ID of the member to be added (required)
      },
      {
          "Member_Account": "jared"
  }]
}
```

For more information, see Adding Group Members.

Webhook for successful team-up: If the maximum number of group members is set when creating a group, the game can start only when all the required number of members are in the group. When the `webhook after a group is full` is received, the game can be started. The following is a sample request URL:

```
https://www.example.com?
SdkAppid=$SDKAppID&CallbackCommand=$CallbackCommand&contenttype=json&ClientIP=$
ClientIP&OptPlatform=$OptPlatform
```

The following is a sample request:

```
{
  "CallbackCommand": "Group.CallbackAfterGroupFull", // Webhook command
  "GroupId": "@TGS#2J4SZEAEL" // Group ID
}
```

For more information, see After a Group Is Full.

Notification on a new member joining the group: When a new player enters the game (group chat), the system sends the `webhook after a user joins a group` to notify other group members. The following is a sample request URL:

```
https://www.example.com?
SdkAppid=$SDKAppID&CallbackCommand=$CallbackCommand&contenttype=json&ClientIP=$
```

```
ClientIP&OptPlatform=$OptPlatform
```

The following is a sample request:

```
{
  "CallbackCommand": "Group.CallbackAfterNewMemberJoin", // Webhook command
  "GroupId" : "@TGS#2J4SZEAEL",
  "Type": "Public", // Group type
  "JoinType": "Apply", // Group joining method: `Apply` (application); `Invited` (i
  "Operator_Account": "leckie", // Operator
  "NewMemberList": [ // List of new members
      {
          "Member_Account": "jared"
      },
      {
          "Member_Account": "tommy
      }
  ]
}
```

Note:

To enable this webhook, you must configure the webhook URL and toggle on the corresponding protocol. For details on the configuration method, see Webhook Configuration. For more information, see After a User Joins a Group.

Leaving a team during a game: When a player leaves the game voluntarily or due to network problems, the server can notify other players in the group that someone leaves the group chat through the `webhook after a user leaves a group` and can also send a message to the player who leaves the game due to network problems. The following is a request URL example for After a User Leaves a Group:

```
https://www.example.com?
SdkAppid=$SDKAppID&CallbackCommand=$CallbackCommand&contenttype=json&ClientIP=$
ClientIP&OptPlatform=$OptPlatform
```

The following is a sample request:

```
{
  "CallbackCommand": "Group.CallbackAfterMemberExit", // Webhook command
  "GroupId": "@TGS#2J4SZEAEL", // Group ID
  "Type": "Public", // Group type
  "ExitType": "Kicked", // Member leaving type: `Kicked` - being kicked out; `Quit`
  "Operator_Account": "leckie", // Operator
  "ExitMemberList": [ // List of members who left the group
      {
          "Member_Account": "jared"
      },
      {
          "Member_Account": "tommy"
```

```
        }
    ]
}
```

Disbanding a group chat after a game ends: After a game ends, the server side can disband the group chat directly. The following is a request URL example for Disbanding a Group:

```
https://adminapisgp.im.qcloud.com/v4/group_open_http_svc/destroy_group?
sdkappid=88888888&identifier=admin&usersig=xxx&random=99999999&contenttype=json
```

The following is a sample request:

```
{
  "GroupId": "@TGS#2J4SZEAEL"
}
```

Audio/Video chat in temporary teams is complex and will be elaborated below.

Getting team information in the background

Getting group details: You can call the server API `Getting Group Profiles` to get group details such as the number of members in the group and the basic information of the group members. You can use the `Filter` field in the request to filter the information to pull. The following is a sample request URL:

```
https://adminapisgp.im.qcloud.com/v4/group_open_http_svc/get_group_info?
sdkappid=88888888&identifier=admin&usersig=xxx&random=99999999&contenttype=json
```

The following is a sample request:

```
{
  "GroupIdList": [ // The list of group IDs specified for the query. This parameter
      "@TGS#1NVTZEAE4",
      "@TGS#1CXTZEAET"
  ]
}
```

For more information, see Getting Group Profiles.

Getting group member details: You can call the API `Getting Group Member Profiles` to get the detailed information of group members (including custom fields). The following is a sample request URL:

```
https://adminapisgp.im.qcloud.com/v4/group_open_http_svc/get_group_member_info?
sdkappid=88888888&identifier=admin&usersig=xxx&random=99999999&contenttype=json
```

The following is a sample request:

```
{
  "GroupId":"@TGS#1NVTZEAE4"  // Group ID (required)
```

```
}
```

In the response, the `MemberNum` field indicates the total number of members in the group, and `AppMemberDefinedData` indicates the custom field information of group members.

Note:

This API does not support audio-video groups (AVChatRoom). For more information, see Getting Group Member Profiles.

Team members getting team information

Members of a team can call the API `Getting Group Member Profiles` to get the detailed information, such as role and status, of other members in the team.

```
GroupGetMemberInfoListParam param = new GroupGetMemberInfoListParam
{
  group_get_members_info_list_param_group_id = "group_id",
  group_get_members_info_list_param_identifier_array = new List<string>
  {
    "user_id"
  }
};
TIMResult res = TencentIMSDK.GroupGetMemberInfoList(param, (int code, string desc,
 // Process the async logic
});
```

Note:

For more information, see Unity - Getting the Profile of a Group Member.

For other group information such as starting a game after a group is full and members joining or leaving a team, see Team-based scenarios. Such information will be notified to the group via callbacks.

For more information about group selection and other group related information, see Group System. For more information about group management, see Console Guide - Group Management.

Filtering sensitive information

Filtering sensitive content is an important feature for games. The implementation scheme is as follows:

1. Bind the webhook before sending a group message.

2. Identify the message type based on the webhook data and deliver the message data to Tencent Cloud Security or another third-party detection service.

3. If the message is an ordinary text message, wait for the detection result of Tencent Cloud Security and return the data packet indicating whether to deliver the message to the Chat backend.

Sample data of the webhook before sending a message:

```
{
    "CallbackCommand": "Group.CallbackBeforeSendMsg", // Webhook command
```

```
    "GroupId": "@TGS#2J4SZEAEL", // Group ID
    "Type": "Public", // Group type
    "From_Account": "jared", // Sender
    "Operator_Account":"admin", // Request initiator
    "Random": 123456, // Random number
    "OnlineOnlyFlag": 1, // The value is `1` if it is an online message and `0` (de
    "MsgBody": [ // Message body. For more information, see the `TIMMessage` messag
        {
            "MsgType": "TIMTextElem", // Text
            "MsgContent":{
                "Text": "red packet"
            }
        }
    ],
    "CloudCustomData": "your cloud custom data"
}
```

Note:

You can identify the message type based on the `MsgType` field in `MsgBody`. For more information on the fields, see Before a Group Message Is Sent.

You can choose to handle a non-compliant message in a specific way, which can be implemented through the data packet in the webhook returned to the Chat backend.

```
{
    "ActionStatus": "OK",
    "ErrorInfo": "",
    "ErrorCode": 0 // Different `ErrorCode` values have different meanings.
}
```

ErrorCode	Description
0	Speaking is allowed, and messages can be delivered normally.
1	Speaking is rejected, and the client returns `10016`.
2	Silent discarding is enabled, and the client returns messages normally.

Note:

You can use them as needed.

Customizing message types (such as props gifting and trading messages)

Game chats require custom messages in addition to simple messages such as text, emoji, and voice messages. Custom messages allow developers to customize the content format of messages to implement more chat room features such as game props gifting and trading.

Tencent Cloud Chat provides nine basic message types: text, emoji, geographical location, image, voice, file, user generated short video (UGSV), system notification, and custom messages. The formats of all messages except custom messages are fixed, and users only need to enter corresponding information. For the detailed descriptions of the message types, see One-to-One Message Types. For more information on the message formats, see Message Formats.

You can send custom messages to users through server APIs Sending One-to-One Messages to One User, Sending One-to-One Messages to Multiple Users, and Sending Ordinary Messages in a Group. The following is a basic request sample for sending an ordinary message in a group:

```
{
    "GroupId": "@TGS#2C5SZEAEF",
    "Random": 8912345, // A random number. If the random numbers of two messages ar
    "MsgBody": [ // Message body, which consists of an element array. For details,
        {
            "MsgType": "TIMTextElem", // Text
            "MsgContent":{
                "Text": "red packet"
            }
        },
        {
            "MsgType": "TIMFaceElem", // Emoji
            "MsgContent":{
            "Index": 6,
            "Data": "abc\\u0000\\u0001"
            }
        }
    ],
    "CloudCustomData": "your cloud custom data",
    "SupportMessageExtension": 0,
}
```

You can modify `CloudCustomData` to define the custom message data (stored on the cloud) and enter the custom message in `MsgBody` (message body) for sending.

Audio/Video chat in teams

Audio/Video chat in games is an important feature. Tencent Cloud Chat apps are equipped with real-time audio/video call features by using Tencent Real-Time Communication (TRTC) and TUICallKit.

Note:

For more information on TUICallKit, see Integration Solution (UI Included) - Audio/Video Call.

Game chat room types

Game chat room types are described as follows:

Type	Characteristics
Channel	A channel usually engages a large number of chat users and does not have a list of fixed members. Users can join and leave a channel at any time. Offline message push is not required.
Live game lobby	A live game lobby usually engages a number of chat users. Users can join and leave the live room at any time. Historical message query is supported.
Team	A team usually engages a small number of chat users, who do not need to be friends with each other. A team is terminated when a game ends. Offline message push is not required.
Friend	One-to-one chat. Chat records are stored. Chat partners can only be friends in the contact list.
Private message	One-to-one chat. Chat partners can be strangers.
Audio-video group	There is no limit on the number of chat users. Users can join and leave an audio-video group at any time.

Chat provides the following types of chat rooms:

Type	Characteristics
Work group (Work)	A work group allows users to join the group by being invited by a friend who is a member of the group. The invitation does not need to be accepted by the invitee or approved by the group owner.
Public group (Public)	A public group allows the group owner to designate group admins. To join the group, a user needs to search for the group ID and send a request, and the request needs to be approved by the group owner or an admin before the user can join the group.
Meeting group (Meeting)	A meeting group allows users to join and leave freely and view historical messages sent before they join the group. Meeting groups are ideal for scenarios that integrate Tencent Real-Time Communication (TRTC), such as audio/video conferencing and online education. This group type is the same as chat room (ChatRoom) in earlier versions.
Audio-video group (AVChatRoom)	An audio-video group allows users to join and exit freely, supports an unlimited number of members, and does not store message history. Audio-video groups can be used with Cloud Streaming Services (CSS) to support on-screen comment chat scenarios.
Community group (Community)	A community group allows users to join and exit freely, supports up to 100,000 members, and stores message history. To join the group, a user needs to search for the group ID and send an application, and the application does not need to be approved by an admin before the user can join the group.

The following provides some sample solutions based on Chat group characteristics, which can be applied to your games as needed.

Type	Solution	Characteristics
Channel and live game lobby	Community group	A community group supports a large number of users and allows users to join and leave freely, without approval.
Friend	One-to-one chat+Permission control (allowing only friends to send messages to each other)	Only friends are allowed to send messages to each other.
Private message	One-to-one chat+Permission control (allowing any two users in the app to send one-to-one messages to each other)	Any two strangers can send messages to each other.
Team-up	Public group (Public) and meeting group (Meeting)	Only team members in the game can enter the group chat. Audio-video chat is supported.
Audio-video group	Audio-video group (AVChatRoom)	There is no limit on the number of group members, and users can join or leave an audio-video group at any time.

Note:

For more information on the permission control of friend and private one-to-one chats, see One-to-One Messaging Permission Control.

For more information on the group system, see Group System.

Similar to the construction scheme of WeChat public account

Last updated：2025-06-20 16:15:49

Our Official channel solution provides subscription-based communication and content distribution services, which is similar to WhatsApp channel's capabilities. Through integrated APIs, enterprises can create and manage verified organizational accounts; Publish articles, rich-media messages, and interactive content; Interact with subscribers and subscribers can also send messages to the Official channel; Monitor operational metrics through REST APIs and third-party callback systems.

Use case :

Brand-owned channels on WhatsApp Business that enable direct customer communication through multimedia messaging (text, images, videos) and service integration.The following are three common use cases of our Official channel:

Real-time Information Broadcast: Direct delivery of news, educational content, and updates to user feeds.

Brand Enhancement：Build recognition and influence through curated multimedia messaging.

Services Integration：Enable transactions (e.g. bank transfers).

The Official channel solution helps businesses generate revenue from their official accounts by optimizing the entire process—from attracting social media users, to establishing direct communication channels, and ultimately converting them into sales revenue.

Overview

An official channel can send broadcast messages to subscribed users and also engage in one-on-one chats with them.

When messages are exchanged, a one-on-one conversation is generated, with the conversationID structured as c2c_officialAccountID.

For management features such as creating an official channel, refer to the server APIs. The IMSDK primarily provides functionalities such as subscribing to an official channel, unsubscribing from an official channel, and retrieving the list of official channels.

Note：

This feature is supported only by the Enhanced edition on v7.6.5011 or later.

Official channel Profile Class Introduction

Attribute	Definition	Description
officialAccountID	official channel ID	The official channel ID must be prefixed with @TOA#_, can be customized, and has a maximum length of 48 bytes.
officialAccountName	official channel name	Maximum Length: 150 bytes (UTF-8 encoded, where 1 Chinese character occupies 3 bytes)
faceUrl	Profile photo of the official channel	Maximum Length: 500 bytes
organization	organization name	Maximum Length: 500 bytes (UTF-8 encoded, where 1 Chinese character occupies 3 bytes)
introduction	Introduction of the official channel	Maximum Length: 400 bytes (UTF-8 encoded, where 1 Chinese character occupies 3 bytes)
customData	custom data	Maximum Length: 3000 bytes
createTime	Creation time of the official channel	Unit: Seconds
subscriberCount	The number of subscribed users	The number of active subscribers to the official channel
subscribeTime	The time when the logged-in user subscribed	Unit: Seconds

Subscribe to an Official channel

To subscribe to an official channel, call the subscribeOfficialAccount method (Java / Swift / Objective-C / C++) and pass the officialAccountID as the parameter.

1. Upon successful subscription, subscribers will receive the onOfficialAccountSubscribed callback notification (Java / Swift / Objective-C / C++) .

2. When the subscribed official channel's profile is modified via server API, subscribers will receive the onOfficialAccountInfoChanged callback notification (Java / Swift / Objective-C / C++) .

3. When the subscribed official channel is deleted via server API, subscribers will receive the onOfficialAccountDeleted callback notification (Java / Swift / Objective-C / C++).

Sample code:

Java

Swift

Objective-C

C++

```
V2TIMManager.getFriendshipManager().subscribeOfficialAccount("official_test", new V
    @Override
    public void onSuccess() {

    }

    @Override
    public void onError(int code, String desc) {

    }
});

V2TIMManager.getFriendshipManager().addFriendListener(new V2TIMFriendshipListener()
    @Override
    public void onOfficialAccountSubscribed(V2TIMOfficialAccountInfo officialAccoun

    }

    @Override
    public void onOfficialAccountDeleted(String officialAccountID) {

    }

    @Override
    public void onOfficialAccountInfoChanged(V2TIMOfficialAccountInfo officialAccou

    }
});
```

```
V2TIMManager.shared.subscribeOfficialAccount(officialAccountID: "officialAccountID"
    print("subscribeOfficialAccount succ")
} fail: { code, desc in
    print("subscribeOfficialAccount fail, \\(code), \\(desc)")
}
V2TIMManager.shared.addFriendListener(listener: self)

func onOfficialAccountSubscribed(officialAccountInfo: V2TIMOfficialAccountInfo) {
```

```
    print("officialAccountInfo:\\(officialAccountInfo.description)");
}

func onOfficialAccountDeleted(officialAccountID: String) {
    print("officialAccountID:\\(officialAccountID)");
}

func onOfficialAccountInfoChanged(officialAccountInfo: V2TIMOfficialAccountInfo) {
    print("officialAccountInfo:\\(officialAccountInfo.description)");
}
```

```
[[V2TIMManager sharedInstance] subscribeOfficialAccount:@"official_test" succ:^ {
    NSLog(@"success");
} fail:^(int code, NSString *desc) {
    NSLog(@"fail, code: %d, msg: %@", code, msg);
}];

[[V2TIMManager sharedInstance] addFriendListener:self];

- (void)onOfficialAccountSubscribed:(V2TIMOfficialAccountInfo *)officialAccountInfo

}

- (void)onOfficialAccountDeleted:(NSString *)officialAccountID {

}

- (void)onOfficialAccountInfoChanged:(V2TIMOfficialAccountInfo *)officialAccountInf

}
```

```
template <class T>
class ValueCallback final : public V2TIMValueCallback<T> {
public:
    using SuccessCallback = std::function<void(const T&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;

    ValueCallback() = default;
    ~ValueCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
      success_callback_ = std::move(success_callback);
      error_callback_ = std::move(error_callback);
    }

    void OnSuccess(const T& value) override {
```

```
      if (success_callback_) {
        success_callback_(value);
      }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
      if (error_callback_) {
        error_callback_(error_code, error_message);
      }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
};

auto callback = new Callback;
callback->SetCallback(
    [=]() {
      delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
      delete callback;
    });
V2TIMManager::GetInstance()->GetFriendshipManager()->SubscribeOfficialAccount(
    "official_test", callback);

class FriendshipListener final : public V2TIMFriendshipListener {
public:
    FriendshipListener() = default;
    ~FriendshipListener() override = default;

    void OnOfficialAccountSubscribed(const V2TIMOfficialAccountInfo &info) override

    }

    void OnOfficialAccountDeleted(const V2TIMString &officialAccountID) override {

    }

    void OnOfficialAccountInfoChanged(const V2TIMOfficialAccountInfo &info) overrid

    }
};

FriendshipListener friendshipListener;
V2TIMManager::GetInstance()->AddFriendshipListener(&friendshipListener);
```

Unsubscribe from an Official channel

To unsubscribe from an official channel, call the unsubscribeOfficialAccount method (Java / Swift / Objective-C / C++) and pass the officialAccountID as the parameter.

After successful unsubscription, the user will receive an onOfficialAccountUnsubscribed callback notification(Java / Swift / Objective-C / C++).

Sample code:

Java

Swift

Objective-C

C++

```
V2TIMManager.getFriendshipManager().unsubscribeOfficialAccount("official_test", new
    @Override
    public void onSuccess() {

    }

    @Override
    public void onError(int code, String desc) {

    }
});

V2TIMManager.getFriendshipManager().addFriendListener(new V2TIMFriendshipListener()
    @Override
    public void onOfficialAccountUnsubscribed(String officialAccountID) {

    }
});
```

```
V2TIMManager.shared.unsubscribeOfficialAccount(officialAccountID: "officialAccountI
    print("unsubscribeOfficialAccount succ")
} fail: { code, desc in
    print("unsubscribeOfficialAccount fail, \\(code), \\(desc)")
}

V2TIMManager.shared.addFriendListener(listener: self)

func onOfficialAccountUnsubscribed(officialAccountID: String) {
    print("officialAccountID:\\(officialAccountID)");
}
```

```
[[V2TIMManager sharedInstance] unsubscribeOfficialAccount:@"official_test" succ:^ {
    NSLog(@"success");
} fail:^(int code, NSString *desc) {
    NSLog(@"fail, code: %d, msg: %@", code, msg);
}];

[[V2TIMManager sharedInstance] addFriendListener:self];

- (void)onOfficialAccountUnsubscribed:(V2TIMOfficialAccountInfo *)officialAccountIn

}
```

```
template <class T>
class ValueCallback final : public V2TIMValueCallback<T> {
public:
    using SuccessCallback = std::function<void(const T&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;

    ValueCallback() = default;
    ~ValueCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
      success_callback_ = std::move(success_callback);
      error_callback_ = std::move(error_callback);
    }

    void OnSuccess(const T& value) override {
      if (success_callback_) {
        success_callback_(value);
      }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
      if (error_callback_) {
        error_callback_(error_code, error_message);
      }
    }

private:
    SuccessCallback success_callback_;
    ErrorCallback error_callback_;
};

auto callback = new Callback;
callback->SetCallback(
    [=]() {
      delete callback;
```

```
    },
    [=](int error_code, const V2TIMString& error_message) {
      delete callback;
    });

V2TIMManager::GetInstance()->GetFriendshipManager()->UnsubscribeOfficialAccount(
    "official_test", callback);

class FriendshipListener final : public V2TIMFriendshipListener {
public:
    FriendshipListener() = default;
    ~FriendshipListener() override = default;

    void OnOfficialAccountUnsubscribed(const V2TIMString &officialAccountID) overri

    }
};

FriendshipListener friendshipListener;
V2TIMManager::GetInstance()->AddFriendshipListener(&friendshipListener);
```

Get Official channel List

Call the getOfficialAccountsInfo interface (Java / Swift / Objective-C / C++) to retrieve the official channel list.
When the officialAccountIDList is empty, it returns the list of subscribed official channels.
When specific official channel IDs are provided in officialAccountIDList, it returns information for those specified official channels.
Sample code:
Java
Swift
Objective-C
C++

```
List<String> officialAccountIDList = new ArrayList<>();
V2TIMManager.getFriendshipManager().getOfficialAccountsInfo(officialAccountIDList,
    @Override
    public void onSuccess(List<V2TIMOfficialAccountInfoResult> v2TIMOfficialAccount

    }

    @Override
    public void onError(int code, String desc) {
```

```
    }
});
```

```
V2TIMManager.shared.getOfficialAccountsInfo(officialAccountIDList: ["officialAccoun
    officialAccountResultList.forEach { item in
        print(item.description)
    }

} fail: { code, desc in
    print("getOfficialAccountsInfo fail, \\(code), \\(desc)")
}
```

```
[[V2TIMManager sharedInstance] getOfficialAccountsInfo:nil succ:^(NSArray<V2TIMOffi
    [self appendString:[NSString stringWithFormat:@"success:%@", resultList]];
} fail:^(int code, NSString *desc) {
    [self appendString:[NSString stringWithFormat:@"fail, code:%d msg:%@",code,des
}];
```

```
template <class T>
class ValueCallback final : public V2TIMValueCallback<T> {
public:
    using SuccessCallback = std::function<void(const T&)>;
    using ErrorCallback = std::function<void(int, const V2TIMString&)>;

    ValueCallback() = default;
    ~ValueCallback() override = default;

    void SetCallback(SuccessCallback success_callback, ErrorCallback error_callback
      success_callback_ = std::move(success_callback);
      error_callback_ = std::move(error_callback);
    }

    void OnSuccess(const T& value) override {
      if (success_callback_) {
        success_callback_(value);
      }
    }
    void OnError(int error_code, const V2TIMString& error_message) override {
      if (error_callback_) {
        error_callback_(error_code, error_message);
      }
    }

private:
    SuccessCallback success_callback_;
```

```
    ErrorCallback error_callback_;
};

V2TIMStringVector officialAccountIDList;

auto callback = new ValueCallback<V2TIMTopicInfoResultVector>{};
callback->SetCallback(
    [=](const V2TIMOfficialAccountInfoResultVector& officialAccountInfoResultList)
      delete callback;
    },
    [=](int error_code, const V2TIMString& error_message) {
      delete callback;
    });

V2TIMManager::GetInstance()->GetFriendshipManager()->GetOfficialAccountsInfo(
    officialAccountIDList, callback);
```

Official channel Messaging

Official channels support two types of messages:

1. Broadcast messages - Sent to all subscribers.

2. One-to-one messages - Private conversations between the official channel and individual subscribers.

Send Broadcast Messages

Use the server-side broadcast message API to send messages to all subscribers of an official channel.

Exchange One-to-One Messages

Subscriber → Official channel:

Use the IMSDK's sendMessage (Java / Swift / Objective-C /C++) with the official channel's officialAccountID as the receiver.

Official channel → Subscriber:

Use the server-side one-to-one message API, specifying:

From_Account: Official channel's officialAccountID

To_Account: Subscriber's userID

Receive Messages

Both message types can be received via the IMSDK's message listener.