TigaseDoc

发行版本 *0.1*

Tigase, Inc.

2022 年 11 月 01 日

Contents

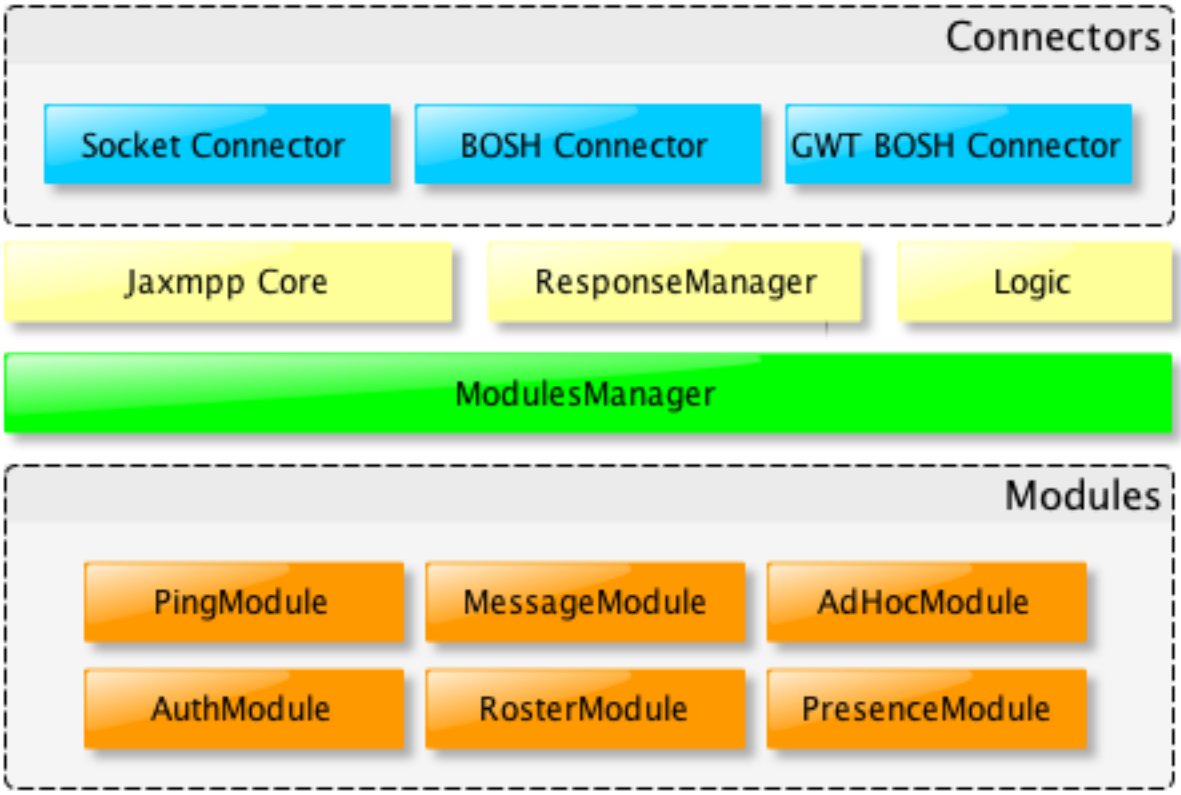
CHAPTER 1

设计与实施

1.1 Tigase Halcyon

Halcyon 是一个多平台可扩展的 XMPP 客户端库

1.2 设计


Connectors
Socket Connector
BOSH Connector
GWT BOSH Connector
Jaxmpp Core
ResponseManager
Logic
ModulesManager
Modules
PingModule
MessageModule
AdHocModule
AuthModule
RosterModule
PresenceModule

CHAPTER 2

快速入门

2.1 最简单的客户端

这是最简单的客户端发送一条消息的示例。

SimplestClient.kt.

```
val halcyon = Halcyon()
halcyon.configuration.let {
    it.setJID("client@tigase.net".toBareJID())
    it.setPassword("secret")
}
halcyon.connectAndWait()

halcyon.request.message {
    to = "romeo@example.net".toJID()
    "body"{
        +"Art thou not Romeo, and a Montague?"
    }
}.send()

halcyon.disconnect()
```

2.2 处理连接状态的变化

我们可以监听连接状态的变化：

```
halcyon.eventBus.register<HalcyonStateChangeEvent>(HalcyonStateChangeEvent.TYPE) {␣
↪stateChangeEvent ->
    println("Halcyon state: ${stateChangeEvent.oldState}->${stateChangeEvent.newState}
↪")
}
```

可用状态：

- `Connecting` - 这个状态意味着，`connect()` 方法被调用，并且与服务器的连接正在进行中。
- `Connected` -连接已完全建立。
- `Disconnecting` - 由于错误或手动断开连接而关闭。
- `Disconnected` - Halcyon 与 XMPP 服务器断开连接，但它仍然处于活动状态。它可能会开始自动重新连接到服务器。
- `Stopped` - Halcyon 已关闭（未激活）。

CHAPTER 3

处理请求

每个模块可以对其他 XMPP 实体执行一些请求，并且（如果是）必须返回 `RequestBuilder` 对象以允许检查请求状态并接收响应。

例如，假设我们要 ping XMPP 服务器（如 XEP-0199 中所述）：

示例 ping 请求和响应。

```
<!-- Client sends: -->
<iq to='tigase.net' id='ping-1' type='get'>
  <ping xmlns='urn:xmpp:ping'/>
</iq>

<!-- Client receives: -->
<iq from='tigase.net' to='client@tigase.net' id='ping-1' type='result'/>
```

Halcyon 中有一个模块 `PingModule` 可以做到这一点：

Kotlin 示例。

```
val pingModule: PingModule = client.modules[PingModule.TYPE]
val request = pingModule.ping("tigase.net".toJID()).send()
```

在这种情况下，方法 `ping()` 返回 `RequestBuilder` 以允许添加结果处理程序、更改默认超时和其他操作。要发送节，您必须调用方法‘send()’。还有可用的方法 `build()` 也创建请求对象，但不发送它。

备注： 在 JVM 上，处理程序的方法将从单独的线程中调用。

以异步方式接收结果的最通用方法是将响应处理程序添加到请求构建器：

```
val client = Halcyon()
val pingModule: PingModule = client.modules[PingModule.TYPE]
pingModule.ping("tigase.net".toJID()).response { result ->
    result.onSuccess { pong ->
        println("Pong: ${pong.time}ms")
    }
    result.onFailure { error ->
        println("Error $error")
    }
}.send()
```

CHAPTER 4

处理事件

Halcyon 是事件驱动的库。这意味着您必须侦听事件以接收消息，对断开连接等做出反应。内部有单个事件总线，您可以向其注册侦听器。库的每个部分（如模块、连接器等）都可能有一组自己的要触发的事件。

注册事件的通用代码：

```
halcyon.eventBus.register<EVENT_TYPE>(EVENT_NAME) { event ->
...
}
```

在 Halcyon 中，事件的名称被定义为每个事件中名为 `TYPE` 的常数变数。

例如：

```
halcyon.eventBus.register<ReceivedXMLElementEvent>(ReceivedXMLElementEvent.TYPE) {␣
↪event ->
    println(" >>> ${event.element.getAsString()}")
}
```

您可以将 EventBus 用于您自己的应用程序。无需注册事件类型。只需创建从 `tigase.halcyon.core.eventbus.Event` 继承的对象并调用方法 `eventbus.fire()`：

```
data class SampleEvent(val sampleData: String) : Event(TYPE){

    companion object {
        const val TYPE = "sampleEvent"
    }
```

(续下页)

(接上页)

```
}

halcyon.eventBus.fire(SampleEvent("test"))
```

CHAPTER 5

模块

Halcyon 包含一组负责执行各种任务的模块：发送和接收消息、身份验证、ping、上传文件、处理名册、存在等。

模块:

5.1 BindModule

资源绑定模块。该模块负责资源绑定，如 RFC 中所述。

5.1.1 特性

- `boundJID` - 在资源绑定过程中包含完整的 JID 绑定，如果客户端未登录和/或未绑定，则为 null。

5.1.2 方法

没有理由在客户端调用此模块的方法。该模块由 Halcyon 库内部使用。

bind(String)

方法准备请求绑定资源。作为参数，它获取建议的资源名称，如果资源名称应由服务器生成，则为 `null`。作为响应，它返回包含完整绑定 JID 的对象 `BindResult` 。

5.2 DiscoveryModule

该模块实现了 XEP-0030: 服务发现.

5.2.1 特性

在这个模块中有几个属性需要设置:

- `clientName` - 客户名称。
- `clientVersion` - 客户端版本。
- `clientCategory` - 客户类别。
- `clientType` - 客户类型。

客户端的类别和类型在 Service Discovery Identities 文档中进行了描述。

5.2.2 事件

ServerFeaturesReceivedEvent

当接收到客户端连接的服务器功能时触发。客户端在登录过程中自动请求服务器功能。

AccountFeaturesReceivedEvent

收到用户帐户功能时触发。客户自动要求这些功能。

5.2.3 方法

info(JID, String)

方法准备 `disco#info` 请求。作为参数，它采用实体的 JID 和节点名称。两者都是可选的。作为响应，返回对象 `Info` 包含实体的 JID、节点名称、身份列表和特征列表。

items(JID, String)

方法准备 disco#items 请求。作为参数，它采用实体的 JID 和节点名称。两者都是可选的。作为响应，返回对象 Items 包含实体的 JID、节点名称和项目列表。

findComponent((Info) → Boolean, (Info) → Unit)

此方法可用于在当前连接的服务器上查找具有特定功能或类型的组件。

作为第一个参数，它需要条件检查器，如果给定 Info 是我们正在搜索的对象，它会检查很多。第二个参数是消费者。

```
findComponent({ candidate ->
                candidate.identities.any { it.type == "mix" }
            }) { result ->
    println("${result.jid}")
}
```

5.3 MessageCarbonsModule

这个模块实现了 XEP-0280: Message Carbons。

5.3.1 事件

Sent

当客户端收到其他实体使用同一帐户发送的消息的副本时触发。事件包含碳化消息。

Received

当客户端收到使用同一帐户的其他实体收到的消息的副本时触发。事件包含碳化消息。

5.3.2 方法

enable()

方法准备请求以在当前会话中启用碳消息。

disable()

方法准备请求以在当前会话中禁用碳消息。

5.4 PingModule

该模块实现了 XEP-0199: XMPP Ping。它允许通过 XML 流 ping XMPP 实体。

5.4.1 已发布的功能

- `urn:xmpp:ping`

5.4.2 方法

ping(JID)

此方法准备 ping 请求。响应对象 `Pong` 包含以毫秒为单位的测量往返时间。

5.5 PresenceModule

用于处理接收到的存在信息的模块。

5.5.1 事件

PresenceReceivedEvent

当客户端收到任何 Presence 节时触发。

字段:

- `jid` -节发件人 JID。
- `stanzaType` - 存在节类型。
- `stanza` - 整个收到的存在节。

ContactChangeStatusEvent

收到存在节但它包含不同的字段集时触发：

- `jid` - 联系人的裸 JID。
- `status` - 联系人设置的人工可读状态。
- `presence` -当前”最佳”存在节，基于存在优先级。
- `lastReceivedPresence` - 刚收到存在节。

请注意，此事件中的 `presence` 可能包含很久以前收到的节。当前事件是由于接收到来自具有较低优先级的实体的存在而引起的。

5.5.2 方法

getPresenceOf(jid: JID)

如果从未从该实体接收到存在，则返回给定实体的存在或 null。

getBestPresenceOf(jid: BareJID)

返回给定裸 JID 的最知名存在。

sendPresence(jid: JID?, type: PresenceType?, show: Show?, status: String)

将存在节发送到特定的 jid。

sendSubscriptionSet(jid: JID, presenceType: PresenceType)

订阅请求快速发送响应的方法。

5.6 RosterModule

该模块实现了 XEP-0060: Publish-Subscribe。它添加了发布-订阅功能。

5.6.1 事件

PubSubEventReceivedEvent

5.6.2 方法

创建 (pubSubJID: JID, node: String, configForm: JabberDataForm? = null)

有趣的订阅 (pubSubJID: JID, node: String, jid: JID)

有趣的 purgeItems(pubSubJID: JID, node: String)

有趣的 retrieveSubscriptions(pubSubJID: JID, node: String)

有趣的 modifySubscriptions(pubSubJID: JID, node: String, subscriptions: List<Subscription>)

有趣的 deleteItem(jid: JID, node: String, itemId: String)

有趣的 retrieveItem(jid: JID, node: String, itemId: String? = null)

有趣的发布 (jid: JID?, node: String, itemId: String?, payload: Element? = null)

有趣的 retrieveAffiliations(jid: JID?, node: String? = null)

5.7 RosterModule

模块负责保存和管理名册项目。

5.7.1 事件

`ItemAdded`

当新项目添加到花名册时触发。

`ItemUpdated`

修改项目时触发。

ItemRemoved

当项目从名册中移除时触发。

5.7.2 方法

addItem(vararg items: RosterItem)

方法准备向名册添加或更新项目的请求。当服务器确认操作时，将触发事件 `ItemAdded` 或者 `ItemUpdated`

deleteItem(vararg jids: BareJID)

方法准备从名册中删除项目的请求。当服务器确认操作时，将触发 `ItemRemoved` 事件。

getAllItems()

方法返回添加已知名册项目。

5.7.3 实现自己的存储

RosterModule 支持名册版本控制，但它需要自定义实现 `RosterStore` 以允许在本地存储名册。默认情况下，Halcyon 有内存名册存储。为此，需要扩展接口 `tigase.halcyon.core.xmpp.modules.roster.RosterStore` 。要使用 `RosterStore` 的自定义实现，只需将其放入 `RosterModule` 中的 `store` 属性即可。请注意，必须在登录前完成。

5.8 SASLModule

模块负责整个客户端认证过程。

5.8.1 特性

- `saslContext` 包含模块的上下文。开始连接时清除上下文。它有几个字段可供阅读：
  - `mechanism` - 使用 SASL 机制，
  - `state` - 身份验证过程的当前状态，
  - `complete` - `true` 如果身份验证过程完成（成功或错误都没有关系）。

5.8.2 事件

SASLStarted

身份验证过程开始时触发。

字段:

- `mechanism` - 使用的 SASL 机制的名称。

SASLSuccess

身份验证成功时触发。

SASLError

身份验证完成但出现错误时触发。

字段:

- `error` - 错误类型的枚举。如果 RFC 会描述 SASL 错误。
- `description` - 人类可读的错误描述（如果由服务器提供）。

5.8.3 方法

没有理由在客户端调用此模块的方法。该模块由 Halcyon 库内部使用。

startAuth()

此方法开始身份验证过程。它不返回 Request 对象。

5.9 VCardModule

此模块允许发布和检索 VCard4，如 XEP-0292 中所述。

5.9.1 特性

- `autoRetrieve` - 如果 `true` 则模块在触发 `VCardUpdatedEvent` 之前自动检索 VCard。默认为 `false`。

5.9.2 事件

`VCardUpdatedEvent`

当收到来自 PEP 的 VCard 更新时触发。包含更新适用的 JID。如果 `autoRetrieve` 设置为 `true` 那么事件将包含当前的 VCard。

5.9.3 方法

`retrieveVCard(jid: BareJID)`

此方法准备检索给定 JID 的 VCard 的请求。结果返回 `VCard` 对象。

`fun publish(vcard: VCard)`

此方法准备发布自己的 vcard 的请求。

5.9.4 使用示例

检索 VCard

```
val vCardModule = halcyon.getModule<VCardModule>(VCardModule.TYPE)!!
vCardModule.retrieveVCard("someone@server.im".toBareJID()).response { result ->
    result.onSuccess {vcard->
        println("""
            Received vcard:
            Name: ${vcard.formattedName}
            Name: ${vcard.structuredName?.given} ${vcard.structuredName?.surname}
            Nick: ${vcard.nickname}
            Birthday: ${vcard.birthday}
            TimeZone: ${vcard.timeZone}
        """.trimIndent())

        println()
        vcard.addresses.forEach {addr->
            println("""
```

(续下页)

(接上页)

```
                ${addr.street}
                ${addr.locality} ${addr.region} ${addr.code}
                ${addr.country}

            """.trimIndent())
        }

    }
    result.onFailure {
        println("Cannot retrieve VCard. Error: $it")
    }
}.send()
```

发布 VCard

发布自己的 vcard 非常简单：

```
val vCardModule = halcyon.getModule<VCardModule>(VCardModule.TYPE)!!
vCardModule.publish(vcard).response { result ->
    result.onSuccess { println("VCard published") }
    result.onFailure { println("VCard NOT published") }
}.send()
```

VCard 对象是可变的，可以编辑。要创建新的 VCard 实例，您可以使用 VCard builder：

```
val vCard = vcard {
    structuredName {
        given = "Alice"
        surname = "Carl"
    }
    nickname = "alice"
    email {
        parameters {
            pref = 1
            +"work"
        }
        text = "alice@organisation.com"
    }
}
```

5.10 BlockingCommandModule

该模块实现了 XEP-0191：阻止命令 和 XEP-0377：垃圾邮件报告

5.10.1 事件

Blocked

新联系人被阻止时触发的事件。它包含几个属性：

- `jid` - 阻止的 JabberID
- `reason` - 阻塞的原因。
- `text` - 可选的人类可读的阻塞原因描述。

Unblocked

当联系人被解除阻止时触发事件。它包含一个属性：

- `jid` - 畅通无阻的 JabberID

UnblockedAll

解锁所有联系人时触发的事件。

```
halcyon.eventBus.register<BlockingCommandEvent>(BlockingCommandEvent.TYPE) { event ->
    when (event) {
        is BlockingCommandEvent.Blocked -> println("$event")
        is BlockingCommandEvent.Unblocked -> println("$event")
        is BlockingCommandEvent.UnblockedAll -> println("All blocked contacts are␣
↪unblocked now!")
    }
}
```

5.10.2 方法

样品：

retrieveList(JID, String)

检索被阻止的联系人列表。它允许获取阻止列表的列表，而无需有关原因的信息。

```
halcyon.getModule<BlockingCommandModule>(BlockingCommandModule.TYPE).retrieveList().
↪response {
    it.onSuccess {
        println("Blocked: $it")
    }
    it.onFailure { println("Oops!") }
}.send()
```

block

屏蔽给定的联系人。

参数:

- `jid` - 要阻止的联系人的 BareJID
- `reason` - 屏蔽的原因（可选，默认值为 `NotSpecified`）
- `text` - 可选的，人类可读的屏蔽描述

```
halcyon.getModule<BlockingCommandModule>(BlockingCommandModule.TYPE)
    .block("spammer@server.com".toBareJID(), Reason.Spam, "It is SPAMMER!!!")
    .response {
        it.onSuccess { println("Done") }
        it.onFailure { println("Oops!") }
    }.send()
```

unblock

取消屏蔽给定的联系人。

参数:

- `jids` - 要取消屏蔽的联系人的 BareJID

备注: 如果 `jids` 为空，则所有被阻止的联系人都将被解除屏蔽!

```
halcyon.getModule<BlockingCommandModule>(BlockingCommandModule.TYPE)
    .unblock("spammer@server.com".toBareJID())
    .response {
```

(续下页)

(接上页)

```
        it.onSuccess { println("Done") }
        it.onFailure { println("Oops!") }
    }.send()
```

unblockAll

取消屏蔽所有被屏蔽的联系人。

```
halcyon.getModule<BlockingCommandModule>(BlockingCommandModule.TYPE)
    .unblockAll().response {
        it.onSuccess { println("Done") }
        it.onFailure { println("Oops!") }
    }.send()
```

5.11 CommandModule

这个模块实现了 XEP-0050: Ad-Hoc Commands.

5.11.1 方法

retrieveCommandList

检索允许在给定 XMPP 实体上执行的命令列表。因为这个命令只是 `DiscoveryModule.items()` 方法的包装，作为响应它返回 `DiscoveryModule.Items` 类。

retrieveCommandInfo

检索有关指定命令的详细信息。因为这个命令只是 `DiscoveryModule.info()` 方法的包装，作为响应它返回 `DiscoveryModule.Info` 类。

executeCommand

此方法对 JabberID 指定的 XMPP 实体执行 Ad-hoc 命令。

参数:

- *jid*‘ - 命令执行者的 Jabber ID,
- `command` -命令名称,
- `form` - 包含数据表单的可选元素,

- action - 命令动作,
- sessionId - 会话标识符，如果命令在会话中执行（标识符由执行器生成）。

作为响应方法返回 AdHocResult 类的对象。它包含结果表单（可选）和命令执行状态。

```
val module = halcyon.getModule<CommandsModule>(CommandsModule.TYPE)
module.executeCommand("responder@domain".toJID(), "configure").response {
    it.onSuccess { result ->
        println("Status: ${result.status}")
        println("Form: ${result.form}")
    }
}.send()
```

如果此命令创建会话，我们可以简单地使用 result 中的数据来执行其中的下一个命令:

```
result.form.getFieldByVar("password").fieldValue = "1234"
module.executeCommand(result.jid, result.node, result.form.createSubmitForm(), null,␣
↪result.sessionId).response {
    it.onSuccess { result ->
        when(result.status){
            Status.Completed -> println("Configured")
            Status.Canceled -> println("Command canceled")
            Status.Executing -> println("Configuration is not finished yet. Next step␣
↪in session ${result.sessionId} is required.")
        }
    }
}.send()
```

上面的示例在第二步中使用默认操作（参数列表中的 null ）。

registerAdHocCommand(command: AdHocCommand)

除了在其他 XMPP 实体上执行命令外，模块还允许注册临时命令，以由其他人在客户端执行。

命令必须实现 AdHocCommand 接口。

```
class TestAdHoc : AdHocCommand {

    override fun isAllowed(jid: BareJID): Boolean = jid == "owner@example.com".
↪toBareJID()

    override val node: String = "command-node-name"
    override val name: String = "Example command"

    override fun process(request: AdHocRequest, response: AdHocResponse) {
```

(续下页)

(接上页)

```
        response.form = createForm()
        response.notes = arrayOf(Note.Info("Everything is OK"))
        response.status = Status.Completed
    }
}
```

以上示例命令只能由 owner@example.com 执行。它甚至隐藏在其他人的命令列表中。

创建的命令必须在 CommandModule 中注册：

```
module.registerAdHocCommand(TestAdHoc())
```

Ad-hoc 命令支持会话。会话允许在会话上下文中存储一些数据并创建多个阶段命令。默认情况下，会话不会自动启动。要在命令中访问会话，请使用方法 request.getSession()。此方法返回当前会话上下文或在必要时创建新的会话上下文。

```
class SessionTestAdHoc : AdHocCommand {

    override fun isAllowed(jid: BareJID): Boolean = jid == "owner@example.com".
↪toBareJID()

    override val node: String = "example-session-adhoc"
    override val name: String = "Example session command"

    override fun process(request: AdHocRequest, response: AdHocResponse) {
        var counter = request.getSession().values["stage"] as Int? ?: 0
        ++counter
        request.getSession().values["stage"] = counter
        if (counter < 3) {
            response.notes = arrayOf(Note.Info("Step $counter"))
            response.actions = arrayOf(Action.Next)
            response.defaultAction = Action.Next
            response.status = Status.Executing
        } else {
            response.notes = arrayOf(Note.Info("Finished"))
            response.status = Status.Completed
        }
    }
}
```

如果响应状态为 Completed 或 Canceled，则会话上下文在命令执行后被销毁。

备注： 请记住，Ad-Hoc Command 的单个实例可能会处理来自许多调用者的调用。

5.12 MUCModule

该模块实现了 XEP-0045：多用户聊天。

5.12.1 事件

MUCModule 中有两种事件：

1. 房间相关事件。
2. 其他事件

目前第二类的唯一事件是 InvitationReceived:

```
halcyon.eventBus.register<MucEvents.InvitationReceived>(MucEvents.TYPE) { event ->
    println("${event.invitation.sender} invites you to room ${event.invitation.
↪roomjid}")
}
```

所有与房间相关的事件都包含 Room 对象，并且都有共同的父对象：

```
halcyon.eventBus.register<MucRoomEvents>(MucRoomEvents.TYPE) { event ->
    when(event){
        is MucRoomEvents.YouJoined -> println("You joined to room ${event.room.
↪roomJID}")
        is MucRoomEvents.OccupantCame -> println("Occupant ${event.nickname} came to $
↪{event.room.roomJID}")
        is MucRoomEvents.OccupantLeave -> println("Occupant ${event.nickname} leaves␣
↪room ${event.room.roomJID}")
        // ...
    }
}
```

InvitationReceived

收到邀请时触发的事件。包含 Invitation 对象。

YouJoined

当服务器成功响应加入请求时触发的事件。

YouLeaved

当您离开房间时触发事件。这可能是您离开请求的确认，或者您被踢出房间。

JoinError

当服务器不接受加入请求时触发事件。

Created

事件通知您加入的房间刚刚创建（通过加入请求）。

OccupantCame

通知新住户加入房间。

OccupantLeave

通知住户离开房间。

OccupantChangedPresence

通知居住者更新了他的存在。

ReceivedMessage

收到来自房间的群聊消息时触发的事件。

5.12.2 方法

join(roomJID, nickname, password)

建立对 MUC Room 的加入请求。

这是一个简单的例子来展示如何加入房间。

```
 halcyon.eventBus.register<MucRoomEvents.YouJoined>(MucRoomEvents.TYPE) {
     println("You joined to room ${it.room.roomJID} as ${it.nickname}")
 }

mucModule.join("coven@chat.shakespeare.lit".toBareJID(), "thirdwitch").send()
```

请注意，由于 MUC 协议的特殊性，加入的确认将作为单独的事件传递。

leave(room: Room)

建立离开 MUC Room 的请求。

destroy(room: Room)

建立房间销毁请求。

invite(room: Room, invitedJid: BareJID, reason: String? = null)

构建中介媒介请求。

inviteDirectly(room: Room, invitedJid: BareJID, reason: String? = null)

构建直接邀请请求。

retrieveRoomConfig(room: Room)

构建检索房间配置请求。作为响应，它返回带有配置的数据表单。

updateRoomConfig(room: Room, form: JabberDataForm)

构建更新房间配置请求。

message(room: Room, msg: String)

构建群聊消息请求。

decline(invitation: Invitation, reason: String? = null)

为收到的邀请构建拒绝请求

accept(invitation: Invitation, nickname: String)

根据收到的邀请建立对 MUC Room 的加入请求。

retrieveAffiliations(room: Room, filter: Affiliation? = null)

建立从 MUC 房间检索隶属关系列表的请求。作为响应，它返回 RoomAffiliation 的集合。

updateAffiliations(room: Room, affiliations: Collection<RoomAffiliation>)

构建更新从属关系列表的请求。

updateRoomSubject(room: Room, subject: String?)

构建设置房间主题的请求。

ping(room: Room)

构建自 ping 请求，如 XEP-0410: MUC Self-Ping (Schrödinger’s Chat) 中所述。

5.12.3 储存

MUCModule 需要 Room Storage 来存储房间数据。默认情况下，Halcyon 带有内存存储。

要构建自己的存储，您必须实现此接口：

```
interface MUCStore {

    fun findRoom(roomJID: BareJID): Room?

    fun createRoom(roomJID: BareJID, nickname: String): Room
}
```

请记住，MUC 协议不适用于移动客户端，因此房间加入的 `state`（在 `Room` 对象中）可能不代表真实状态。例如，在重新连接后，客户端可能会保持 `Joined` 状态，但服务器会收到有关断开连接的信息并将住户从房间中移除。要检查是否需要重新加入，请使用 `ping(room)` 函数。

5.13 SIMSModule

该模块部分实现了 XEP-0385: Stateless Inline Media Sharing (SIMS)。它描述了文件共享元数据。

`SIMSModule` 不是 Halcyon 架构意义上的模块。这是帮助检索和生成 SIMS 结构的扩展集合。

5.13.1 方法

`getReferenceOrNull()`

如果存在，方法返回 `Reference` 对象。

`getMediaSharingFileOrNull()`

此方法（扩展 `Reference`）返回共享文件的详细信息。

`createFileSharingReference()`

此方法使用共享文件详细信息创建完整的 `Reference` 对象。

5.14 MAMModule

该模块实现了 XEP-0313：消息存档管理。

5.14.1 事件

`MAMMessageEvent`

当接收到 `query()` 方法的每个结果时，将触发此事件。

此事件包含以下字段：

- `resultStanza` - 收到的整个消息节（对查询的响应），
- `queryId` - 查询标识符，
- `id` - 结果标识符，
- `forwardedStanza` - 包含来自存档和原始接收时间戳的节的查询结果

5.14.2 方法

query(to: BareJID? = null, node: String? = null, rsm: RSM.Query? = null, with: String? = null, start: Long? = null, end: Long? = null)

从存档中检索聊天记录的主要方法。

参数:

- to - MAM 组件的 JID。如果为 null，则使用用户服务器的默认 MAM 组件，
- node - 节点的名称，
- rsm - 结果集管理对象，
- with - 对话者的名字（JID）
- start, end - 按接收日期过滤消息的时间戳

以上所有参数都可能是 null。

retrievePreferences()

检索 MAM 首选项。

作为回应，您将获得 Preferences 对象，其中包含:

- default - 消息归档的默认行为: Always, Never, Roster。
- always - BareJID 的集合，与他们的对话将始终被存档。
- never - 永远不会存档与之对话的 BareJID 集合。

updatePreferences(preferences: Preferences)

更新 MAM 首选项。

5.14.3 用法

当客户端与服务器建立连接时，它应该向服务器询问与连接到同一帐户的其他客户端交换的所有消息。

可以通过向服务器询问自上次收到消息以来的所有消息来完成:

```
val mamModule = halcyon.getModule<MAMModule>(MAMModule.TYPE)
fun ask(q: RSM.Query? = null) {
    mamModule.query(
        with = "someone@tigase.org",
        start = lastReceivedMessageTimestatmp,
        rsm = q
```

(续下页)

(接上页)

```
    ).response { res ->
        res.onSuccess {
            println("Complete: ${it.complete} :: ${it.rsm}")

            if (!it.complete) {
                ask(RSM.Query(after = it.rsm!!.last))
            }
        }
    }.consume { forwardedStanza: ForwardedStanza<Message> ->
        if (forwardedStanza.stanza.body != null) println(
            "FROM MAM | ${forwardedStanza.resultId} $
↪{timestampToISO8601(forwardedStanza.timestamp!!)} ${forwardedStanza.stanza.from}: $
↪{forwardedStanza.stanza.body}"
        )
    }.send()
}
ask()
```

因为 MAM 服务器已经定义了返回消息的最大数量，所以我们必须询问直到查询完成。在示例中，它是通过使用填充的结果集管理对象反复执行方法 `ask()` 来完成的。`RSM.Query(after = it.rsm!!.last)` 意味着结果中必须只包含位于当前接收到的消息包中最后一个消息标识符之后的消息。

请注意，方法 `query()` 的参数 "with" 是可选的，因此您可以询问自特定时间以来与特定 JID 交换的所有消息，或者您可以询问存储在存档中的所有消息。

从存档中检索消息的第二种方法是询问位于特定消息标识符之前或之后的消息。

```
val mamModule = halcyon.getModule<MAMModule>(MAMModule.TYPE)
fun ask(q: RSM.Query? = null) {
    mamModule.query(
        with = "someone@tigase.org",
        rsm = q
    ).response { res ->
        res.onSuccess {
            println("Complete: ${it.complete} :: ${it.rsm}")

            if (!it.complete) {
                ask(RSM.Query(after = it.rsm!!.last))
            }
        }
    }.consume { forwardedStanza: ForwardedStanza<Message> ->
        if (forwardedStanza.stanza.body != null) println(
            "FROM MAM | ${forwardedStanza.resultId} $
↪{timestampToISO8601(forwardedStanza.timestamp!!)} ${forwardedStanza.stanza.from}: $
```

(续下页)

(接上页)

```
↪{forwardedStanza.stanza.body}"
        )
    }.send()
}
ask(RSM(after = lastKnownMessageId))
```

要获取 MAM 相关的消息 ID，您必须使用 `getStanzaIDBy()` 函数（作为参数，您必须输入自己的帐户名称）：

```
val mamMessageId = message.getStanzaIDBy("myaccount@tigase.org".toBareJID())
```

当其被服务器推送到客户端时，当''以正常方式''接收消息时很有用。

当使用方法 `query()`（在消费者中）接收到消息时，标识符在 `ForwardedStanza` 中给出：

```
val mamMessageId = forwardedStanza.resultId
```

您可以使用相同的机制来加载历史记录中的旧消息（客户端尚未加载）：

```
mamModule.query(
    with = "someone@tigase.org",
    rsm = RSM.Query(before = "2753e4a8-9150-4e34-8757-4cd5e8419159", max = 20)
).response { res ->
    res.onSuccess {
        println("Complete: ${it.complete} :: ${it.rsm}")
    }
}.consume { forwardedStanza: ForwardedStanza<Message> ->
    println(
        "FROM MAM | ${forwardedStanza.resultId} ${
            timestampToISO8601(
                forwardedStanza.timestamp!!
            )
        } ${forwardedStanza.stanza.from}: ${forwardedStanza.stanza.body}"
    )
}.send()
```

在上面的示例中，客户端要求位于在消息 `2753e4a8-9150-4e34-8757-4cd5e8419159` 之前历史记录中的 20 条消息。

备注： 消息存档返回请求的消息量。并非每条消息都可能包含要显示的正文。其中一些消息可能仅包含对消息读取或其他控制命令的确认。

CHAPTER 6

Jabber 数据表

XEP-0004 中描述了 Jabber 数据表单。数据表单在 XEP 中未描述的所有工作流中都很有用。例如服务配置或搜索结果。

6.1 使用表单

要访问接收表单的字段，我们必须创建 `JabberDataForm` 对象：

```
val form = JabberDataForm(formElement)
```

其中 `formElement` 是 `<x xmlns='jabber:x:data'>` XML 元素的表示。

每个表单可能具有以下属性：

- `type` - form type,
- `title` -可选的表格标题，
- `description` - 可选的，人类可读的，形式的描述。

字段由 `var` 名称标识。每个字段可能有 字段类型 （它是可选的）。

让我们看一下，如何列出所有具有值的字段：

```
val form = JabberDataForm(element)
println("Title: ${form.title}")
println("Description: ${form.description}")
```

(续下页)

(接上页)

```
println("Type: ${form.type}")
println("Fields:")
form.getAllFields().forEach {
    println(" - ${it.fieldName}: ${it.fieldType}  (${it.fieldLabel})  == ${it.
↪fieldValue}")
}
```

要按名称获取字段，只需使用：

```
val passwordField = form.getFieldByVar("password")
```

这些字段的值可以修改：

```
passwordField.fieldValue = "******"
```

完成所有表单修改后，有时我们需要将填写好的表单发回。准备提交的表单有单独的方法：

```
val formElement = form.createSubmitForm()
```

此方法准备 `<x xmlns='jabber:x:data'>` 类型为 `submit` 的 XML 元素，并且所有字段都从不必要的元素（如描述或标签）中清除。它只留下带有名称和值的简单字段。

6.2 创建表单

我们可以创建新表单，设置标题和描述，并添加字段：

```
val form = JabberDataForm.create(FormType.Form)
form.addField("username", FieldType.TextSingle)
form.addField("password", FieldType.TextPrivate).apply {
    fieldLabel = "Enter password"
    fieldDesc = "Password must contain at least 8 characters"
    fieldRequired = true
}
```

要在不清理表单的情况下获取包含表单的 XML 元素，只需使用：

```
val formElement = form.element
```

6.3 多值响应

有一个包含许多字段集的表单变体。这种形式已经声明了一组具有名称的列和一组包含具有之前声明的名称的字段的项目。

此示例显示如何显示具有值的所有字段：

```
val form = JabberDataForm(element)
val columns = form.getReportedColumns().mapNotNull { it.fieldName }
columns.forEach { print("$it;  ") }
println()
println("------------")
form.getItems().forEach { item ->
    columns.forEach { col -> print("${item.getValue(col).fieldValue};  ") }
    println()
}
```

创建多值形式也很简单。首先，我们必须设置报告列的列表，因为当添加新项目时，会根据声明的列检查字段名称。

```
val form = JabberDataForm.create(FormType.Result)
form.title = "Bot Configuration"
form.setReportedColumns(listOf(Field.create("name", null), Field.create("url", null)))
form.addItem(
    listOf(Field.create("name").apply { fieldValue = "Comune di Verona - Benvenuti␣
↪nel sito ufficiale" },
           Field.create("url").apply { fieldValue = "http://www.comune.verona.it/" })
)
form.addItem(
    listOf(Field.create("name").apply { fieldValue = "Universita degli Studi di␣
↪Verona - Home Page" },
           Field.create("url").apply { fieldValue = "http://www.univr.it/" })
)
```