openwechat

发行版本 *v1*

eatMoreApple

2023 年 09 月 25 日

Contents

CHAPTER 1

Bot 对象

Bot 对象负责处理网络请求和消息回调以及登录登出的用户行为，一个 Bot 对应一个登录的微信号。

1.1 创建 Bot 对象

在登录微信之前需要创建一个 Bot 对象。

```
bot := openwechat.DefaultBot()
```

使用默认的构造方法 DefaultBot 来创建一个 Bot 对象。

1.2 登陆二维码回调

但仅仅是这样我们依然无法登录，我们平常登录微信都需要用手机扫描二维码登录，所以我们得知道需要扫描哪张二维码，最后还需要为它绑定一个登录二维码的回调函数。

```
// 注册登陆二维码回调
bot.UUIDCallback = openwechat.PrintlnQrcodeUrl
```

PrintlnQrcodeUrl 这个函数做的事情很简单，就是将我们需要扫码登录的二维码链接打印打控制台上，这样我们就知道去扫描哪张二维码登录了。

可以自定义 UUIDCallback 来实现自己的逻辑。

如：将登录的二维码打印到控制台。

```
package main

import (
        "fmt"
        "github.com/skip2/go-qrcode"
        "github.com/eatMoreApple/openwechat"
)

func ConsoleQrCode(uuid string) {
        q, _ := qrcode.New("https://login.weixin.qq.com/l/"+uuid, qrcode.Low)
        fmt.Println(q.ToString(true))
}

func main() {
        bot := openwechat.DefaultBot()
        bot.UUIDCallback = ConsoleQrCode
        bot.Login()
}
```

虽然最终打印的结果肉眼看上去有点不尽人意，但手机也还能够识别...

1.3 登录

1.3.1 扫码登录

上面的准备工作做完了，下面就可以登录，直接调用 `Bot.Login` 即可。

```
bot.Login()
```

`Login` 方法会阻塞当前 goroutine，直到登录成功或者失败。

登录会返回一个 `error`，即登录失败的原因。

1.3.2 热登录

每次执行普通登录都需要扫码，在调试一些功能的时候需要反复编译，这样会很麻烦。

热登录可以只用扫码一次，后面在单位时间内重启程序也不会再要求扫码

```
// 创建热存储容器对象
reloadStorage := openwechat.NewFileHotReloadStorage("storage.json")

defer reloadStorage.Close()
```

(续下页)

(接上页)

```
// 执行热登录
bot.HotLogin(reloadStorage)
```

HotLogin 需要接受一个热存储容器对象来调用。热存储容器用来保存登录的会话信息，本质是一个接口类型

我们第一次进行热登录的时候，因为我们的热存储容器是空的，所以这时候会发生错误。

我们只需要在 HotLogin 增加一个参数，让它在失败后执行扫码登录即可

```
bot.HotLogin(reloadStorage, openwechat.NewRetryLoginOption())
```

当扫码登录成功后，会将会话信息写入到热存储容器中，下次再执行热登录的时候就会从热存储容器中读取会话信息，直接登录成功。

```
// 热登陆存储接口
type HotReloadStorage io.ReadWriter
```

NewFileHotReloadStorage 简单实现了该接口，它采用文件的方式存储会话信息。

实现这个接口，来定义你自己的存储方式。

1.3.3 免扫码登录

目前热登录有一点缺点就是它的有效期很短（具体多久我也不知道）。

我们平常在 pc 上登录微信的时候，通常只需要登录一次，第二次就会在微信上有一个确认登录的按钮，点击确认就会往手机上发送一个确认登录的请求，这样就可以免扫码登录了。

openwechat 也提供了这样的功能。

```
bot.PushLogin(storage HotReloadStorage, opts ...openwechat.BotLoginOption) error
```

PushLogin 需要传入一个热存储容器，和一些可选参数。

HotReloadStorage 跟上面一样，用来保存会话信息，必要参数。

openwechat.BotLoginOption 是一个可选参数，用来设置一些额外的行为。

目前有下面几个可选参数：

```
//  NewRetryLoginOption 登录失败后进行扫码登录
func NewRetryLoginOption() BotLoginOption
```

注意：如果是第一次登录，PushLogin 一定会失败的，因为我们的 HotReloadStorage 里面没有会话信息，你需要设置失败会进行扫码登录。

```
bot := openwechat.DefaultBot()
reloadStorage := openwechat.NewFileHotReloadStorage("storage.json")
defer reloadStorage.Close()
err = bot.PushLogin(reloadStorage, openwechat.NewRetryLoginOption())
```

这样当第一次登录失败的时候，会自动执行扫码登录。

扫码登录成功后，会自动保存会话信息到 HotReloadStorage，下次登录就可以直接使用 PushLogin 了，就会往手机上发送确认登录的请求。

1.4 扫码回调

在 pc 端微信上我们打开手机扫码进行登录的时候，只扫描二维码，但不点击确认，微信上也能够显示当前扫码用户的头像，并提示用户登录确认。

通过对 bot 对象绑定扫码回调即可实现对应的功能。

```
bot.ScanCallBack = func(body openwechat.CheckLoginResponse) { fmt.
↪Println(string(body)) }
```

用户扫码后，body 里面会携带用户的头像信息。

注：绑定扫码回调须在登录前执行。

CheckLoginResponse 是一个 []byte 包装类型，扫码成功后可以通过该类型获取用户的头像信息。

```
type CheckLoginResponse []byte

func (c CheckLoginResponse) Avatar() (string, error)
```

1.5 登录回调

对 bot 对象绑定登录

```
bot.LoginCallBack = func(body openwechat.CheckLoginResponse) {
                fmt.Println(string(body))
                // to do your business
}
```

登录回调的参数就是当前客户端需要跳转的链接，用户可以不用关心它。(其实可以拿来做一些骚操作)

登录回调函数可以当做一个信号处理，表示当前扫码登录的用户已经确认登录。

1.6 桌面模式

DefaultBot 默认是与网页版微信进行交互，部分用户的网页版 wx 可能已经被限制登录了。

这时候可以尝试使用桌面模式进行登录。

```
bot := openwechat.DefaultBot(openwechat.Desktop)
```

别的逻辑不用改，直接在创建 bot 的时候加一个参数就行了。

如果桌面模式还登录不上，请检查你的微信号是不是刚刚申请。

1.7 消息处理

在用户登录后需要实时接受微信发送过来的消息。

很简单，给 BOT 对象绑定一个消息回调函数就行了。

```
// 注册消息处理函数
bot.MessageHandler = func(msg *openwechat.Message) {
                if msg.IsText() && msg.Content == "ping" {
                        msg.ReplyText("pong")
                }
}
```

所有接受的消息都通过 Bot.MessageHandler 来处理。

基于这个回调函数，可以对消息进行多样化处理

```
dispatcher := openwechat.NewMessageMatchDispatcher()

// 只处理消息类型为文本类型的消息
dispatcher.OnText(func(ctx *openwechat.MessageContext){
        msg := ctx.Message
        fmt.Println("Text: ", msg.Content)
        msg.ReplyText("hello")
})

// 注册消息回调函数
bot.MessageHandler = dispatcher.AsMessageHandler()
```

openwechat.DispatchMessage 会将消息转发给 dispatcher 对象处理

1.7.1 MessageMatchDispatcher

构造方法

```
openwechat.NewMessageMatchDispatcher()
```

注册消息处理函数

```
// 注册消息处理函数
func (m *MessageMatchDispatcher) RegisterHandler(matchFunc matchFunc, handlers ...
↪MessageContextHandler)


// 消息匹配函数
type matchFunc func(*Message) bool


// 消息处理函数
type MessageContextHandler func(ctx *MessageContext)
```

matchFunc：接受当前收到的消息对象，并返回 bool 值，返回 true 则表示处理当前的消息

RegisterHandler：接受一个 matchFunc 和不定长的消息处理函数，如果 matchFunc 返回为 true，则表示运行对应的处理函数组。

OnText

注册处理消息类型为文本类型的消息

```
func (m *MessageMatchDispatcher) OnText(handlers ...MessageContextHandler)
```

OnImage

注册处理消息类型为图片类型的消息

```
func (m *MessageMatchDispatcher) OnImage(handlers ...MessageContextHandler)
```

OnVoice

注册处理消息类型为语言类型的消息

```
func (m *MessageMatchDispatcher) OnVoice(handlers ...MessageContextHandler)
```

更多请点击查看源码

1.8 获取登录后的用户

```
self, err := bot.GetCurrentUser()
```

注：该方法在登录成功后调用

详见 Self 对象

1.9 阻塞主程序

```
bot.Block()
```

该方法会一直阻塞，直到用户主动退出或者网络请求发生错误。

1.10 控制 Bot 存活

判断当前的 `Bot` 是否存活。

```
func (b *Bot) Alive() bool
```

当返回为 `true` 则表示 `Bot` 存活。

如何控制 `Bot` 存活呢?

```
ctx, cancel := context.WithCancel(context.Background())

bot := openwechat.DefaultBot(openwechat.WithContext(ctx))
```

`WithContext` 接受一个 `context.Context` 对象，当 `context` 对象被取消时，`Bot` 也会被取消。

当前我们也可以调用 `bot.Logout` 来主动退出当前的 `Bot`，当 `Bot` 退出后，`bot.Alive()` 会返回 `false`。

CHAPTER 2

用户

抽象的用户结构: 好友群组公众号

```
type User struct {
        Uin                int
        HideInputBarFlag   int
        StarFriend         int
        Sex                int
        AppAccountFlag     int
        VerifyFlag         int
        ContactFlag        int
        WebWxPluginSwitch  int
        HeadImgFlag        int
        SnsFlag            int
        IsOwner            int
        MemberCount        int
        ChatRoomId         int
        UniFriend          int
        OwnerUin           int
        Statues            int
        AttrStatus         int
        Province           string
        City               string
        Alias              string
        DisplayName        string
```

(续下页)

(接上页)

```
        KeyWord           string
        EncryChatRoomId   string
        UserName          string
        NickName          string
        HeadImgUrl        string
        RemarkName        string
        PYInitial         string
        PYQuanPin         string
        RemarkPYInitial   string
        RemarkPYQuanPin   string
        Signature         string

        MemberList Members

        Self *Self
}
```

User 结构体的属性，部分信息可以通过它的英文名知道它所描述的意思。

其中要注意的是 UserName 这个属性。

UserName 是当前会话唯一的身份标识，且仅作用于当前会话。下次登录该属性值则会被改变。

不同用户的 UserName 的值是不一样的，可以通过该字段来区分不同的用户。

2.1 获取用户唯一标识

不同于 UserName，ID 是用户的唯一标识，且不会随着登录而改变。

```
func (u *User) ID() string
```

2.2 获取头像

下载群聊、好友、公众号的头像，具体哪种类型根据当前 User 的抽象类型来判断

```
func (u *User) SaveAvatar(filename string) error
```

2.3 详情

获取制定用户的详细信息, 返回新的用户对象

```
func (u *User) Detail() (*User, error)
```

2.4 判断是否为好友

```
func (u *User) IsFriend() bool
```

2.5 判断是否为群组

```
func (u *User) IsGroup() bool
```

2.6 判断是否为公众号

```
func (u *User) IsMP() bool
```

2.7 当前登录用户

当前扫码登录的用户对象

Self 拥有上面 User 的全部属性和方法

通过调用 bot.GetCurrentUser 来获取

```
self, err := bot.GetCurrentUser()
```

2.7.1 获取当前用户的所有的好友

```
firends, err := self.Friends() // self.Friends(true)
```

Friends: 可接受 bool 值来判断是否获取最新的好友

2.7.2 获取当前用户的所有的群组

```
groups, err := self.Groups()  // self.Groups(true)
```

注：群组列表只显示手机端微信：通讯录：群聊里面的群组，若想将别的群组加入通讯录，点击群组，设置为保存到通讯录即可（安卓机）

如果需要获取不在通讯录里面的群组，则需要收到来自该群组的消息，然后再次调用 `self.Groups()` 来获取

`Groups`：可接受 `bool` 值来判断是否获取最新的群组

2.7.3 获取当前用户所有的公众号

```
mps, err := self.Mps()  // self.Mps(true)
```

`Mps`：可接受 `bool` 值来判断是否获取最新的公众号

2.7.4 获取文件传输助手

```
fh, err := self.FileHelper()
```

2.7.5 发送文本给好友

```
func (s *Self) SendTextToFriend(friend *Friend, text string) (*SentMessage, error)
```

```
firends, err := self.Friends()

if err != nil {
    return
}

if friends.Count() > 0 {
    self.SendTextToFriend(friends.First(), "hello")
    // 或者
    // friends.First().SendText("hello")
}
```

返回的 `SentMessage` 对象可用于消息撤回

2.7.6 发送图片消息给好友

```
// 确保获取了有效的好友对象
img, _ := os.Open("your file path")
defer img.Close()
self.SendImageToFriend(friend, img)
// 或者
// friend.SendImage(img)
```

2.7.7 发送文件给好友

```
file, _ := os.Open("your file path")
defer file.Close()
self.SendFileToFriend(friend, file)
// 或者
// friend.SendFile(img)
```

2.7.8 给好友设置备注

```
self.SetRemarkNameToFriend(friend, "你的备注")
// 或者
// friend.SetRemarkName("你的备注")
```

2.7.9 发送文本消息给群组

```
self.SendTextToGroup(group, "hello")
// group.SendText("hello")
```

2.7.10 发送图片给群组

```
img, _ := os.Open("your file path")
defer img.Close()
self.SendImageToGroup(group, img)
// group.SendImage(img)
```

2.7.11 发送文件给群组

```
file, _ := os.Open("your file path")
defer file.Close()
self.SendFileToGroup(group, file)
// group.SendFile(file)
```

2.7.12 消息撤回

```
sentMesaage, _ := friend.SendText("hello")
self.RevokeMessage(sentMesaage)
// sentMesaage.Revoke()
```

只要是 openwechat.SentMessage 对象都可以在 2 分钟之内撤回

2.7.13 消息转发给多个好友

```
sentMesaage, _ := friend.SendText("hello")

self.ForwardMessageToFriends(sentMesaage, friends1, friends2)

// sentMesaage.ForwardToFriends(friends1, friends2)
```

2.7.14 转发消息给多个群组

```
sentMesaage, _ := friend.SendText("hello")

self.ForwardMessageToGroups(sentMesaage, group1, group2)

// sentMesaage.ForwardToGroups(friends1, friends2)
```

2.7.15 拉多个好友入群

```
self.AddFriendsIntoGroup(group, friend1, friend2) // friend1, friend2 为不定长参数

// group.AddFriendsIn(friend1, friend2)
```

最好自己是群主，这样成功率会高一点。

2.7.16 拉单个好友进多个群

```
self.AddFriendIntoManyGroups(friend, group1, group2) // group1, group2 为不定长参数

// friend.AddIntoGroup(group1, group2)
```

2.7.17 从群聊中移除用户

```
member, err := group.Members()

self.RemoveMemberFromGroup(group, member[0], member[1])

// group.RemoveMembers(member[:1])
```

注：这个接口已经被微信官方禁用了，现在已经无法使用。

2.8 好友

2.8.1 好友列表

```
type Friends []*Friend
```

获取当前用户的好友列表

```
firends, err := self.Friends()
```

注：此时获取到的 `friends` 为好友组，而非好友。好友组是当前 wx 号所有好友的集合。

统计好友个数

```
firends.Count() // => int
```

自定义条件查找好友

```
func (f Friends) Search(limit int, condFuncList ...func(friend *Friend) bool)␣
↪(results Friends)
```

- limit：限制查找的个数
- condFuncList：不定长参数，查找的条件，必须全部满足才算匹配上
- results：返回的满足条件的好友组

```
// 例：查询昵称为eatmoreapple的1个好友
sult := firends.Search(1, func(friend *openwechat.Friend) bool {return friend.
↪NickName == "eatmoreapple"})
```

根据昵称查找好友

```
func (f Friends) SearchByNickName(limit int, nickName string) (results Friends)
```

- limit：为限制好友查找的个数
- nickname：查询指定昵称的好友
- results：返回满足条件的好友组

根据备注查找好友

```
func (f Friends) SearchByRemarkName(limit int, remarkName string) (results Friends)
```

- limit：为限制好友查找的个数
- remarkname：查询指定备注的好友
- results：返回满足条件的好友组

群发文本消息

```
func (f Friends) SendText(text string, delay ...time.Duration) error
```

- text：文本消息的内容
- delay：每次发送消息的间隔（发送消息过快可能会被 wx 检测到，最好加上间隔时间）

群发图片

```
func (f Friends) SendImage(file io.Reader, delay ...time.Duration) error
```

- file: io.Reader 类型。
- delay: 每次发送消息的间隔（发送消息过快可能会被 wx 检测到，最好加上间隔时间）

群发文件

```
func (f Friends) SendFile(file io.Reader, delay ...time.Duration) error
```

- file: io.Reader 类型。
- delay: 每次发送消息的间隔（发送消息过快可能会被 wx 检测到，最好加上间隔时间）

2.8.2 单个好友

获取好友头像

```
friend.SaveAvatar("avatar.png")
```

发送文本信息

```
friend.SendText("hello")
```

发送图片信息

```
img, _ := os.Open("your image path")

defer img.Close()

friend.SendImage(img)
```

发送文件信息

```
file, _ := os.Open("your file path")

defer file.Close()

friend.SendFile(file)
```

设置备注信息

```
friend.SetRemarkName("你的备注")
```

拉该好友进群

```
friend.AddIntoGroup(group)
```

2.9 群组

2.9.1 群组列表

```
type Groups []*Group
```

获取所有的群聊

```
groups, err := self.Groups()
```

注：该方法在用户成功登陆之后调用

统计群聊个数

```
groups.Count() // => int
```

自定义条件查找群聊

```
func (g Groups) Search(limit int, condFuncList ...func(group *Group) bool) (results
↪Groups)
```

- limit：限制查找的个数
- condFuncList：不定长参数，查找的条件，必须全部满足才算匹配上
- results：返回的满足条件的群聊

根据群名查找群聊

```
func (g Groups) SearchByNickName(limit int, nickName string) (results Groups)
```

- limit：限制查找的个数
- nickName：群名称
- results：返回的满足条件的群聊

群发文本

```
func (g Groups) SendText(text string, delay ...time.Duration) error
```

- text：文本消息的内容
- delay：每次发送消息的间隔（发送消息过快可能会被 wx 检测到，最好加上间隔时间）

群发图片

```
func (g Groups) SendImage(file io.Reader, delay ...time.Duration) error
```

- file：io.Reader 类型。
- delay：每次发送消息的间隔（发送消息过快可能会被 wx 检测到，最好加上间隔时间）

群发文件

```
func (g Groups) SendFile(file io.Reader, delay ...time.Duration) error
```

- file: io.Reader 类型。
- delay: 每次发送消息的间隔（发送消息过快可能会被 wx 检测到，最好加上间隔时间）

2.9.2 单个群聊

获取群聊头像

```
group.SaveAvatar("group.png")
```

获取所有的群员

```
members, err := group.Members()
```

发送文本信息

```
group.SendText("hello")
```

发送图片信息

```
img, _ := os.Open("your image path")

defer img.Close()

group.SendImage(img)
```

发送文件消息

```
file, _ := os.Open("your file path")

defer file.Close()

group.SendFile(file)
```

拉好友进群

```
group.AddFriendsIn(friend)
```

CHAPTER 3

消息

3.1 接受消息

被动接受的消息对象，由微信服务器发出

消息对象通过绑定在 bot 上的消息回调函数获取

```
bot.MessageHandler = func (msg *openwechat.Message) {
if msg.IsText() && msg.Content == "ping" {
msg.ReplyText("pong")
}
}
```

以下简写为 msg

3.1.1 消息内容

```
msg.Content // 获取消息内容
```

通过访问 Content 属性可直接获取消息内容

由于消息分为很多种类型，它们都共用 Content 属性。一般当消息类型为文本类型的时候，我们才会去访问 Content 属性。

3.1.2 消息类型判断

下面的判断消息类型的方法均返回 `bool` 值

文本消息

```
msg.IsText()
```

图片消息

```
msg.IsPicture()
```

位置消息

```
msg.IsLocation()
```

语音消息

```
msg.IsVoice()
```

是否为好友添加请求

```
msg.IsFriendAdd()
```

名片消息

```
msg.IsCard()
```

视频消息

```
msg.IsVideo()
```

是否被撤回

```
msg.IsRecalled()
```

系统消息

```
msg.IsSystem()
```

收到微信转账

```
msg.IsTransferAccounts()
```

发出红包 (自己发出)

```
msg.IsSendRedPacket()
```

收到红包

```
msg.IsReceiveRedPacket()
```

但是不能领取!

判断是否为拍一拍

```
msg.IsIsPaiYiPai() // 拍一拍消息
msg.IsTickled()
```

判断是否拍了拍自己

```
msg.IsTickledMe()
```

判断是否有新人加入群聊

```
msg.IsJoinGroup()
```

3.1.3 获取消息的发送者

```
sender, err := msg.Sender()
```

如果是群聊消息，该方法返回的是群聊对象 (需要自己将 `User` 转换为 `Group` 对象)

3.1.4 获取消息的接受者

```
receiver, err := msg.Receiver()
```

3.1.5 获取消息在群里面的发送者

```
sender, err := msg.SenderInGroup()
```

获取群聊中具体发消息的用户，前提该消息必须来自群聊。

3.1.6 是否由自己发送

```
msg.IsSendBySelf()
```

3.1.7 是否为拍一拍

```
msg.IsTickled()
```

3.1.8 消息是否由好友发出

```
msg.IsSendByFriend()
```

3.1.9 消息是否由群聊发出

```
msg.IsSendByGroup()
```

3.1.10 回复文本消息

```
msg.ReplyText("hello")
```

3.1.11 回复图片消息

```
img, _ := os.Open("your file path")
defer img.Close()
msg.ReplyImage(img)
```

3.1.12 回复文件消息

```
file, _ := os.Open("your file path")
defer file.Close()
msg.ReplyFile(file)
```

3.1.13 获取消息里的其他信息

名片消息

```
card, err := msg.Card()
```

该方法调用的前提为 `msg.IsCard()` 返回为 `true`

名片消息可以获取该名片中的微信号

```
alias := card.Alias
```

`card` 结构

```
// 名片消息内容
type Card struct {
XMLName                 xml.Name `xml:"msg"`
ImageStatus             int      `xml:"imagestatus,attr"`
Scene                   int      `xml:"scene,attr"`
```

(续下页)

(接上页)

```
Sex                       int      `xml:"sex,attr"`
Certflag                  int      `xml:"certflag,attr"`
BigHeadImgUrl             string   `xml:"bigheadimgurl,attr"`
SmallHeadImgUrl           string   `xml:"smallheadimgurl,attr"`
UserName                  string   `xml:"username,attr"`
NickName                  string   `xml:"nickname,attr"`
ShortPy                   string   `xml:"shortpy,attr"`
Alias                     string   `xml:"alias,attr"` // Note: 这个是名片用户的微信号
Province                  string   `xml:"province,attr"`
City                      string   `xml:"city,attr"`
Sign                      string   `xml:"sign,attr"`
Certinfo                  string   `xml:"certinfo,attr"`
BrandIconUrl              string   `xml:"brandIconUrl,attr"`
BrandHomeUr               string   `xml:"brandHomeUr,attr"`
BrandSubscriptConfigUrl string   `xml:"brandSubscriptConfigUrl,attr"`
BrandFlags                string   `xml:"brandFlags,attr"`
RegionCode                string   `xml:"regionCode,attr"`
}
```

获取已撤回的消息

```
revokeMsg, err := msg.RevokeMsg()
```

该方法调用成功的前提是 `msg.IsRecalled()` 返回为 `true`

撤回消息的结构

```
type RevokeMsg struct {
SysMsg     xml.Name `xml:"sysmsg"`
Type       string   `xml:"type,attr"`
RevokeMsg struct {
OldMsgId   int64  `xml:"oldmsgid"`
MsgId      int64  `xml:"msgid"`
Session    string `xml:"session"`
ReplaceMsg string `xml:"replacemsg"`
} `xml:"revokemsg"`
}
```

3.1.14 同意好友请求

```
friend, err := msg.Agree()
// msg.Agree("我同意了")
```

返回的 friend 即刚添加的好友对象

该方法调用成功的前提是 `msg.IsFriendAdd()` 返回为 `true`

3.1.15 设置为已读

```
msg.AsRead()
```

该当前消息设置为已读

3.1.16 设置消息的上下文

用于多个消息处理函数之间的通信，并且是协程安全的。

设置值

```
msg.Set("hello", "world")
```

获取值

```
value, exist := msg.Get("hello")
```

3.2 已发送消息

已发送消息指当前用户发送出去的消息

每次调用发送消息的函数都会返回一个 `SentMessage` 对象

如

```
sentMsg, err := msg.ReplyText("hello") // 通过回复消息获取
// sentMsg, err := friend.SendText("hello") // 向好友对象发送消息获取
// and so on
```

3.2.1 撤回消息

撤回刚刚发送的消息，撤回消息的有效时间为 2 分钟，超过了这个时间则无法撤回

```
sentMsg.Revoke()
```

3.2.2 判断是否可以撤回

```
sentMsg.CanRevoke()
```

3.2.3 转发给好友

```
sentMsg.ForwardToFriends(friend1, friend2)
```

将刚发送的消息转发给好友

3.2.4 转发给群聊

```
sentMsg.ForwardToGroups(group1, group2)
```

将刚发送的消息转发给群聊

3.3 Emoji 表情

openwechat 提供了微信全套 `emoji` 表情的支持

`emoji` 表情全部维护在 `openwechat.Emoji` 结构体上

emoji 表情可以通过发送 `Text` 类型的函数发送

如

```
firend.SendText(openwechat.Emoji.Doge) // 发送狗头表情
msg.ReplyText(openwechat.Emoji.Awesome) // 发送666的表情
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search