HotGo--WebSocket，消息处理，心跳，在线用户

本篇文章记录了关于 HotGo 中使用 WebSocket 的使用，WebSocket 的创建，事件的监听，心跳，在线用户，涉及到前端和后端的代码，整体上都是 go 和 js 的代码，vue3 鲜有涉及。

后端连接准备

后端需要先注册路由，使用 gin 的中间件做登录验证，然后等待前端来访问，现在就从路由看起

注册路由

```
// internal/router/websocket.go
group.GET("/", websocket.WsPage)
```

注册接收事件

```
// internal/router/websocket.go
websocket.RegisterMsg(websocket.EventHandlers{
    "ping":                  common.Site.Ping,       // 心跳
    "join":                  common.Site.Join,       // 加入组
    "quit":                  common.Site.Quit,       // 退出组
    "admin/monitor/trends":  admin.Monitor.Trends,  // 后台监控，动态数据
    "admin/monitor/runInfo": admin.Monitor.RunInfo, // 后台监控，运行信息
})
```

因为 WebSocket 也有很多类事件，比如心跳事件，监控数据事件等，需要一

个字段来区分，这里时用 map 来注册不同的事件类型，通过 event 这个 key 来区分，比如 { event: ping }, 代表的时候心跳事件。

```
// internal/websocket/init.go
// WsPage ws入口
func WsPage(r *ghttp.Request) {
    upGrader := websocket.Upgrader{
        ReadBufferSize:  1024,
        WriteBufferSize: 1024,
        CheckOrigin: func(r *http.Request) bool {
            return true
        },
    }
    conn, err := upGrader.Upgrade(r.Response.Writer, r.Request, nil)
    if err != nil {
        return
    }
    currentTime := uint64(gtime.Now().Unix())
    client := NewClient(r, conn, currentTime)
    go client.read()
    go client.write()
    // 用户连接事件
    clientManager.Register <- client
}
```

接收到 WebSocket 请求以后，通过 WsPage 函数将协议升级为 WebSocket 协议，并创建一个 client 对象，然后通过两个 goroutine 协程 read 和 write 来处理这个 client 的 socket 消息。

```
// internal/websocket/client.go
// read 函数
for {
    _, message, err := c.Socket.ReadMessage()
    if err != nil {
        return
    }
    // 处理消息
    handlerMsg(c, message)
}
```

```

read 函数中有一个 for 循环，一直在读消息，读到之后就转到 handlerMsg 中处理来自客户端的消息。

```
// internal/websocket/client.go
// write 函数
for {
    select {
    case <-c.closeSignal:
        g.Log().Infof(mctx, "websocket client quit, user:%+v", c.User)
        return
    case message, ok := <-c.Send:
        if !ok {
            // 发送数据错误 关闭连接
            g.Log().Warningf(mctx, "client write message, user:%+v", c.User)
            return
        }
        _ = c.Socket.WriteJSON(message)
    }
}
```

write 函数中，通过 for select + channel 的组合来建立了一个消息管道，等待要发送的消息，如果要发送消息，就往这个管道中发送就可以了，这个管道就是 c.Send, 是一个 chan 的指针类型。

## 前端创建 WebSocket 实例，并连接

前端创建 WebSocket 实例，连接的地址后面带上 token，做认证用，token 是登录时获取的 jwt，里面有部分用户身份信息。连接之后，收发数据使用的是 WebSocket 协议。无论是 HTTP 还是 WebSocket，都是基于 tcp 来传输数据的，这一点大家要清楚。

注意在建立 WebSocket 连接之前，客户端会先发送一个 HTTP 请求到服务器，请求升级到 WebSocket 协议。这个请求会包含一些特殊的头部字段，例如 Upgrade 和 Sec-WebSocket-Key。服务器收到请求后，会进行验证并返回响应。如果验证成功，服务器会返回一个 101 状态码，并包含一些用于建立 WebSocket 连接的参数，例如 Sec-WebSocket-Accept。客户端收到响应后，就完成了 WebSocket 协议的握手升级。
```

```
// src\utils\websocket\index.ts
socket = new
WebSocket(`${useUserStore.config?.wsAddr}?authorization=${useUserSt
ore.token}`);
init();
```

在上面的 init 的函数中，注册了 socket 的几个关键回调函数，来处理业务逻辑，onopen、onmessage、onclose、onerror。

心跳包

作者在 index.ts 中创建了一个 heartCheck 对象，用于管理心跳，每隔 5s 发送一个心跳，心跳的内容只有一个简单的json内容：{"event":"ping"}，同样，返回的结构也很简单 {"event":"ping","code":0,"timestamp":1715670733}。这个过程中还有一些细节的控制，比如发出心跳后，5s 内服务端没有返回，就关闭 WebSocket，关闭之后，就会尝试重连等。

```
const heartCheck = {
  timeout: 5000,
  // ...
  start: function () {
    // eslint-disable-next-line @typescript-eslint/no-this-alias
    const self = this;
    clearTimeout(this.timeoutObj);
    clearTimeout(this.serverTimeoutObj);
    this.timeoutObj = setTimeout(function () {
      socket.send(
        JSON.stringify({
          event: SocketEnum.EventPing,
        })
      );
      self.serverTimeoutObj = setTimeout(function () {
        console.log('[WebSocket] 关闭服务');
        socket.close();
      }, self.timeout);
```

```
    }, this.timeout);
  },
};
```

服务端在接收到心跳 ping 以后，也做出了简单的回应，注意这个事件是在 controller 层开始接收处理的，通过 SendSuccess 调用到 websocket 模块

```
// internal/controller/websocket/handler/common/site.go
func (c *cSite) Ping(client *websocket.Client, req *websocket.WRequest)
{
    websocket.SendSuccess(client, req.Event)
}
```

```
// SendSuccess 发送成功消息
func SendSuccess(client *Client, event string, data ...interface{}) {
    d := interface{}(nil)
    if len(data) > 0 {
        d = data[0]
    }
    client.SendMsg(&WResponse{
        Event:     event,
        Data:      d,
        Code:      gcode.CodeOK.Code(),
        Timestamp: gtime.Now().Unix(),
    })
    before(client)
}

func before(client *Client) {
    client.Heartbeat(uint64(gtime.Now().Unix()))
}
```

在 Heartbeat 函数中更新了，当前 client 的心跳时间，以便下次计算超时，下线等功能。

```
// internal/router/websocket.go
// 启动websocket监听
websocket.Start()

// internal/websocket/init.go
// Start 启动
func Start() {
    go clientManager.start()
    go clientManager.ping()
    g.Log().Debug(mctx, "start websocket..")
}

// internal/websocket/client_manager.go
// 定时任务，清理超时连接
_, _ = gcron.Add(mctx, "*/30 * * * * *", func(ctx context.Context) {
    manager.clearTimeoutConnections()
})
```

我们可以看到，这里也是通过一个 goroutine 启动了一个异步任务，在 client_manager.go 中，每隔 30s 清理一次已经超时的 client，将超时的连接关闭。

在线用户

在系统监控-在线用户，可以看到在线的用户列表，那么这是怎么实现的呢，在线用户的数据其实都保存在 ClientManager 中，在请求时，直接返回即可，具体代码：

```
// internal/websocket/init.go
// Start 启动
func Start() {
    go clientManager.start()
    go clientManager.ping()
    g.Log().Debug(mctx, "start websocket..")
}
```

```
```

```
// internal/websocket/client_manager.go
// 管道处理程序
func (manager *ClientManager) start() {
    for {
        select {
        case conn := <-manager.Register:
            // 建立连接事件
            manager.EventRegister(conn)
        case login := <-manager.Login:
            // 用户登录
            manager.EventLogin(login)
        case conn := <-manager.Unregister:
            // 断开连接事件
            manager.EventUnregister(conn)
        case message := <-manager.Broadcast:
            // 全部客户端广播事件
            clients := manager.GetClients()
            for conn := range clients {
                conn.SendMsg(message)
            }
        // 其他广播事件...
}

```

在这个 start 函数中，会注册监听很多事件，比如 Register，Unregister，就会影响到 client 的增加与减少，这些用户数据就是通过 ClientManager 中的 Clients 获取的。

```
// internal/websocket/client_manager.go
// GetClients 获取所有客户端
func (manager *ClientManager) GetClients() (clients map[*Client]bool) {
    clients = make(map[*Client]bool)
    manager.ClientsRange(func(client *Client, value bool) (result bool) {
        clients[client] = value
        return true
    })
    return
}

```

更多关于 clients 的操作都可以在 client_manager.go 中找到。

另外，除了这些，WebSocket 还可以向用户发送消息，就像上面代码中提到的广播事件。

原文链接: https://juejin.cn/post/7368692841297477686