GoChatRoom

- 웹소켓을 사용해서 go언어에서 서버를 띄우고 Docker까지 올렸었던 프로젝트
- 해당 프로젝트는 최종적으론 회사에서 거절당했으므로 개인 프로젝트가 되버렸습니다.

Session

```go
type chatSession struct {
    conn    *websocket.Conn
    inbox   chan []byte
    port    string
    key     string
    address string

    // 컨탠트
    stop           bool
    uid            string
    usernick       string
    currentChannel *ChatRoom
    currentMatch   *ChatRoom

    m sync.Mutex
}
```

```go
func (p *chatSession) removeChannel() {
    p.m.Lock()
    defer p.m.Unlock()
    p.currentChannel = nil
}

func (p *chatSession) removeMatch() {
    p.m.Lock()
    defer p.m.Unlock()
    p.currentMatch = nil
}

func (p *chatSession) closeSession() {
    p.conn.Close()

    if p.currentChannel != nil {
        p.currentChannel.LeaveUser(p.uid)
    }

    if p.currentMatch != nil {
        p.currentMatch.LeaveMatchUser(p.uid)
    }

    p.stop = true
    Sessions.DeleteSession(p.uid)
    log.Printf("세션이 종료되었습니다. uid : %s\n", p.uid)
}
```

```go
func (n *chatSession) read() {
    defer n.closeSession()

    for {
        m := PacketData{}
        err := n.conn.ReadJSON(&m)

        if err != nil {
            break
        }

        handleMessage(&m, n)
    }
}

func (n *chatSession) write() {
    defer n.closeSession()
    for {
        m, ok := <-n.inbox

        if ok == false {
            break
        }

        log.Printf("유저에게 보내는 패킷 : %s\n", string(m))
        n.conn.WriteMessage(websocket.TextMessage, m)
    }
}
```

Session

```go
func initSession(conn *websocket.Conn, address string, uid string, nickname string) *chatSession {
    key := fmt.Sprintf("%s", uid)

    println(key)

    loggined := FindSession(Sessions, key)
    if loggined != nil {
        loggined.closeSession()
    }

    n := &chatSession{
        conn:    conn,
        inbox:   make(chan []byte),
        address: address,
        key:     key,

        usernick: nickname,
        uid:      uid,
    }

    Sessions.RegisterSession(n)
    go n.read()
    go n.write()

    return n
}
```

Session

- 연결을 위한 소켓이 필요하지만 여기서는 웹소켓으로 대체되었습니다.
- Go 명령어로 유사 스레드를 두개를 생성해서 소켓을 받거나 보낼때 까지 기달리게 해줍니다. 여기서 스레드가 너무 많아지는 문제가 발생하지만, Go언어 공식 문서를 살펴보면 해당 Go 키워드는 몇백만개를 만들어도 상관 없다는 표시가 있었으므로, 이런식으로 채용했습니다
- 원래라면 복잡하게 헤더와 내용물을 포장하거나 뜯어내는 작업을 해야했지만, 당시 바빴던 관계로 Json을 그대로 주고 받았습니다
- 해당 방에서 처리는 되겠지만, 일단 추가 제거 작업만큼은 락을 걸어 데이터를 안전하게 보호하고 바로 풀어주는 작업을 했습니다.

SessionManager

```go
func (m *SessionManager) RegisterSession(newSession *chatSession) bool {
    m.m.Lock()
    defer m.m.Unlock()

    m.session[newSession.uid] = newSession
    return true
}
```

```go
var MainContext context.Context
var Sessions *SessionManager

type SessionManager struct {
    session map[string]*chatSession
    m       sync.Mutex
}

func InitSessionManager() {
    MainContext = context.Background()

    Sessions = &SessionManager{
        session: make(map[string]*chatSession),
    }
}

func GetEverySession(m *SessionManager) map[string]*chatSession {
    return m.session
}

func FindSession(m *SessionManager, uid string) *chatSession {
    m.m.Lock()
    defer m.m.Unlock()

    session, exist := m.session[uid]

    if exist == false {
        return nil
    }

    return session
}

func (m *SessionManager) DeleteSession(uid string) bool {
    m.m.Lock()
    defer m.m.Unlock()

    _, exist := m.session[uid]

    if exist == true {
        delete(m.session, uid)
        return true
    }

    return false
}
```

SessionManager

- 세션을 관리 등록하는 클래스입니다
- 이전 프로젝트 중 라이브 5분전에 세션간 문제가 생기는 점이 발생했는데 알고보니 이 락을 오히려 걸지 않아서 생긴 문제였습니다. 이 이후로, 데이터의 등록 해제만큼은 바로 lock을 걸어서 처리해주는 습관이 생기게 됐습니다.

Rest

```go
func Start(targetPort int) {
    router := mux.NewRouter()
    port = fmt.Sprintf(":%d", targetPort)

    templates = template.Must(template.ParseGlob(templateDir + "html/*.gohtml"))

    mime.AddExtensionType(".js", "application/javascript; charset=utf-8")
    mime.AddExtensionType(".css", "text/css; charset=utf-8")

    headersOk := handlers.AllowedHeaders([]string{"Origin", "Accept", "Content-Type", "text", "X-Requested-With", "Access-Control-Allowed-Origin"})
    originsOk := handlers.AllowedOrigins([]string{"*"})
    methodsOk := handlers.AllowedMethods([]string{"GET", "HEAD", "POST", "PUT", "OPTIONS"})

    mydir, err := os.Getwd()
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(mydir)

    router.PathPrefix("/asset").Handler(http.StripPrefix("/asset", http.FileServer(http.Dir("./Rest/static"))))
    router.PathPrefix("/css").Handler(http.StripPrefix("/css", http.FileServer(http.Dir("./Rest/static/css"))))
    router.PathPrefix("/js").Handler(http.StripPrefix("/js", http.FileServer(http.Dir("./Rest/static/js"))))

    router.HandleFunc("/passreset", reqResetPass)
    router.HandleFunc("/node", nodes).Methods("GET", "POST")
    router.HandleFunc("/serverStatus", serverStatus).Methods("GET")
    router.HandleFunc("/roomlist", roomlist).Methods("GET")
    //router.HandleFunc("/google_authtest", GoogleAuthRes)
    router.HandleFunc("/ws", socket.Upgrade).Methods("GET")

    fmt.Printf("서버 활성화 http://127.0.0.1%s\n", port)
    http.ListenAndServe(port, handlers.CORS(headersOk, originsOk, methodsOk)(router))
```

Rest

- 웹에서 진입하고 라우터를 설정하는 곳입니다.
- IOCP서버했던 것 처럼 클라이언트가 진입하면 받아들일 IOObject가 필요하듯이 여기서는 post를 웹소켓으로 이어질 부분이 필요합니다. /ws에서 socket.Upgrade 함수를 사용하여 받아드릴 준비를 하면됩니다.

Listener

```go
var upgrader = websocket.Upgrader{}

func Upgrade(rw http.ResponseWriter, r *http.Request) {
    connIp, _ := utils.ExtractClientAddressFromRequest(r)

    // 여기서 클라가 세션이랑 토큰값 던져주면 redis로 체크해서 true 아니면 false로 리턴할것
    uid := r.URL.Query().Get("uid")
    nick := r.URL.Query().Get("nick")

    upgrader.CheckOrigin = func(r *http.Request) bool {
        return true
    }

    conn, err := upgrader.Upgrade(rw, r, nil)
    utils.HandleError(err)
    initSession(conn, connIp, uid, nick)

    log.Printf("유저가 접속했습니다. uid = %s, nick = %s\n", uid, nick)
}
```

Listener

- Post 요청이 도착하여 웹소켓으로 변환 될 곳입니다.
- 이 위치에서 InitSession으로 세션이 생성됩니다.
- 여기서 왜 i가 아니라 I인지 궁금해 하실텐데, 이는 GO언어의 고유한 특징때문입니다. 이를 통해서 public과 private이 나뉘게 됩니다.

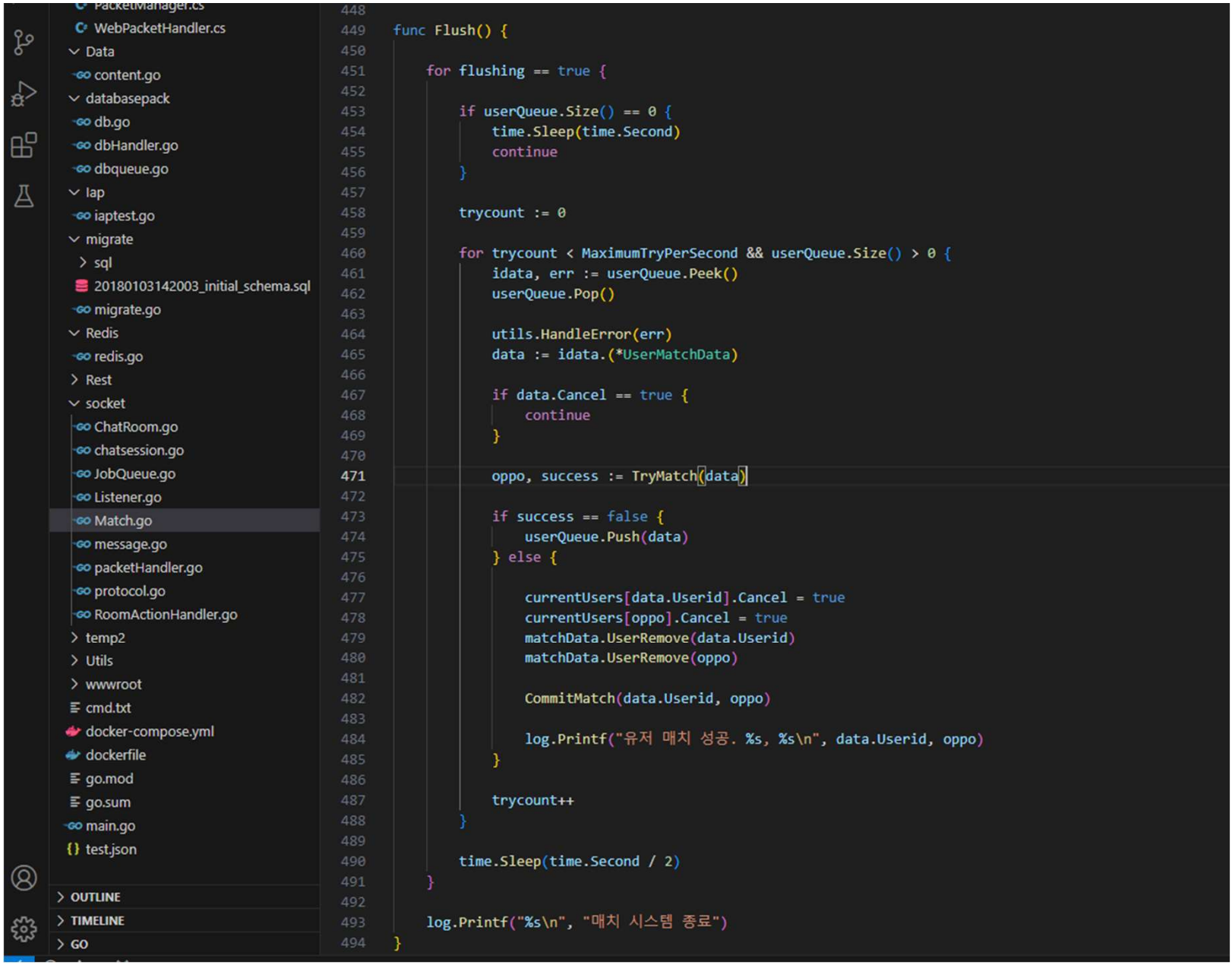
Main

```go
func main() {
    // 포트번호

    c := make(chan os.Signal, 2)
    //m := make(chan bool)

    signal.Notify(c, os.Interrupt, syscall.SIGINT, syscall.SIGTERM)

    socket.InitSessionManager()
    socket.InitChannel()
    socket.InitPacketHandler()
    socket.InitMatching()

    go socket.Flush()
    go rest.Start(8090)

    <-c
    //cancel()
    socket.KillFlush()

    //databasepack.CloseDB()
    socket.StopEveryChannel()

    fmt.Println("Shutdown complete")
    os.Exit(0)
```

Main

- 기본 프로젝트의 진입점입니다. Go언어에서의 채널 기능을 이용하여, Join의 역할을 대신하게 하고, Socket의 초기화를 시켜주는 함수를 작성하고, 비동기 함수를 Queue형식으로 처리하는 유사 스레드 하나와, restful의 형을 갖추는 유사 스레드를 하나 배치시켜서 병렬 처리를 시행하게 했습니다.

Socket




```
448
449 func Flush() {
450
451     for flushing == true {
452
453         if userQueue.Size() == 0 {
454             time.Sleep(time.Second)
455             continue
456         }
457
458         trycount := 0
459
460         for trycount < MaximumTryPerSecond && userQueue.Size() > 0 {
461             idata, err := userQueue.Peek()
462             userQueue.Pop()
463
464             utils.HandleError(err)
465             data := idata.(*UserMatchData)
466
467             if data.Cancel == true {
468                 continue
469             }
470
471             oppo, success := TryMatch(data)
472
473             if success == false {
474                 userQueue.Push(data)
475             } else {
476
477                 currentUsers[data.Userid].Cancel = true
478                 currentUsers[oppo].Cancel = true
479                 matchData.UserRemove(data.Userid)
480                 matchData.UserRemove(oppo)
481
482                 CommitMatch(data.Userid, oppo)
483
484                 log.Printf("유저 매치 성공. %s, %s\n", data.Userid, oppo)
485             }
486
487             trycount++
488         }
489
490         time.Sleep(time.Second / 2)
491     }
492
493     log.Printf("%s\n", "매치 시스템 종료")
494 }
```

Socket

- Flush 함수는 실질적으로는 매칭이 주 된 작업이였습니다. 유저 간의 매칭을 구현을 했어야 했는데 단순한 큐로 먼저하는 규칙을 만들고, 그 다음에 TryMatch로 그래프 형식으로 매칭을 해 주었습니다.
- 매칭이 성공을 했다면 신청한 유저를 제거하고, 매칭된 유저의 매칭요청을 취소하고 일종의 룸을 만들어서 해당 두 유저를 같은 룸에 넣는 작업을 시켰습니다.

Queue

- Flush 함수는 실질적으로는 매칭이 주 된 작업이였습니다. 유저 간의 매칭을 구현을 했어야 했는데 단순한 큐로 먼저하는 규칙을 만들고, 그 다음에 TryMatch로 그래프 형식으로 매칭을 해 주었습니다.
- 매칭이 성공을 했다면 신청한 유저를 제거하고, 매칭된 유저의 매칭요청을 취소하고 일종의 룸을 만들어서 해당 두 유저를 같은 룸에 넣는 작업을 시켰습니다.

JobTimerQueue

```go
var ErrQueueEmpty = errors.New("큐가 비었습니다.")

type IJob interface {
    Dispatch()
    PrintName()
}

type JobTimerElem struct {
    Action   IJob
    ExecTick int
    Cancel   bool
}

type JobTimerQueue struct {
    items []JobTimerElem
    s     sync.Mutex
}

func (p *JobTimerQueue) Size() int {

    return len(p.items)
}

func Peek(p *JobTimerQueue) (*JobTimerElem, error)
    if p.Size() == 0 {
        temp := new(JobTimerElem)
        return temp, ErrQueueEmpty
    }

    return &p.items[0], nil
}
```

```go
func (p *JobTimerQueue) Push(currentTick int, after int, action IJob
    jobElem := JobTimerElem{
        ExecTick: currentTick + after,
        Action:   action,
        Cancel:   false,
    }

    p.s.Lock()
    defer p.s.Unlock()

    p.items = append(p.items, jobElem)
    num := p.Size() - 1

    for num > 0 {
        next := (num - 1) / 2

        if p.items[num].ExecTick <= p.items[next].ExecTick {
            break
        }

        temp := p.items[num]
        p.items[num] = p.items[next]
        p.items[next] = temp

        num = next
    }
}
```

```go
func (p *JobTimerQueue) Pop() (JobTimerElem, error) {
    if p.Size() == 0 {
        return JobTimerElem{}, ErrQueueEmpty
    }

    res := p.items[0]

    lastIndex := p.Size() - 1
    p.items[0] = p.items[lastIndex]
    p.items = p.items[:lastIndex]

    lastIndex--

    now := 0
    for now > 0 {
        left := now*2 + 1
        right := now*2 + 2

        next := now

        if left <= lastIndex && p.items[next].ExecTick > p.items[left].ExecTick {
            next = left
        }

        if right <= lastIndex && p.items[next].ExecTick > p.items[right].ExecTick {
            next = right
        }

        if next == now {
            break
        }

        temp := p.items[now]
        p.items[now] = p.items[next]
        p.items[next] = temp

        now = next
    }

    return res, nil
}
```

JobTimerQueue

```go
func (p *JobTimerQueue) Flush(name string, currentTick int) {
    defer p.s.Unlock()

    for {
        p.s.Lock()

        {
            if p.Size() == 0 {
                break
            }

            log.Printf("현재룸 틱 : %s, %d", name, p.Size())
            peekJob, err := Peek(p)
            utils.HandleError(err)

            if peekJob.ExecTick > currentTick {
                break
            }

            job, _ := p.Pop()
            job.Action.Dispatch()
        }

        p.s.Unlock()
    }
}
```

JobTimerQueue

- 일감을 쌓아놨다가 예약을 해서 쓸때 필요한 경우가 있었습니다. 원래라면 하나의 객체내에서 Flush 명령을 내리는 것이 더 좋지만, go 특성을 이용하여 각각의 방마다 Flush를 시켰습니다.
- 이 객체는 각각의 방마다 예약 일감을 실행하는 것에 집중을 하였습니다. Flush주기는 1초당 한번이였습니다. (메신저 서버임을 감안)
- 그러기 위해선 일단 Priority_queue를 직접 구현할 필요가 있었습니다. 알고리즘 문제해결 전략 책에서 본 내용을 그대로 사용해서 queue를 가까운 순차적으로 나열하여 구현할 수 있었습니다

GameRoom

```go
var rooms *Rooms
var roomnumber int = 0
var ErrUserNotFound = errors.New("해당 유저를 찾을 수 없습니다.")

const (
    CHANNAL_COUNT = 10
    MAX_PEOPLE    = 500
    MAX_CHATLOG   = 100
)

const (
    TYPE_CHANNEL = 0
    TYPE_MATCH   = 1
    TYPE_MEETING = 2
    TYPE_TEMP    = 3
    TYPE_LARGE   = 4
)

type Rooms struct {
    ChannalRooms map[string]*ChatRoom
    MatchRooms   map[string]*ChatRoom
    PrivateRooms map[string]*ChatRoom

    // 혹시 모르니까?...
    m sync.Mutex
}

func InitChannel() {
    if rooms != nil {
        return
    }

    rooms = &Rooms{
        ChannalRooms: make(map[string]*ChatRoom),
        MatchRooms:   make(map[string]*ChatRoom),
        PrivateRooms: make(map[string]*ChatRoom),
    }

    BootingChannel()
}
```

```go
type ChatRoom struct {
    UserList    map[string]*chatSession
    PendingList JobTimerQueue
    m           sync.Mutex

    Roomnumber    int
    RoomHost      *chatSession
    RoomUID       string
    RoomName      string
    RoomType      int
    Password      string
    MaximumPeople int

    ChatLog  utils.Queue
    tick     int
    StopRoom bool
}

type ChattingLog struct {
    UserUid  string `json:"uid"`
    UserName string `json:"username"`
    Chat     string `json:"chat"`
}

func tickRoom(r *ChatRoom) {
    for {
        if r.StopRoom == true {
            break
        }

        r.PendingList.Flush(r.RoomName, r.tick)
        time.Sleep(time.Second)
        r.tick++
    }
}
```

```go
func (r *ChatRoom) EnterUser(newUser *chatSession) {
    if newUser == nil {
        log.Printf("유저가 없습니다.")
        return
    }

    {
        registerAction := &RegisterUserAction{
            Room:       r,
            TargetUser: newUser,
        }

        r.pushAction(0, registerAction)
    }

    {
        var res S_CHANNELENTER
        res.JoinSuccess = true
        res.Roomname = r.RoomName
        var userlist []data.UserData
        for _, val := range r.UserList {

            user := data.UserData{
                UserUid:  val.uid,
                Username: val.use
            }

            userlist = append(use
        }

        res.UserData = userlist

        targetnoti := &UniCastAct
            Room:       r,
            protocol:   S_Channel
            packet:     res,
            targetUser: newUser,
        }

        r.pushAction(0, targetnot
    }

    {
        userdata := data.UserData{
            UserUid:  newUser.uid,
            Username: newUser.usernick,
        }

        var result []data.UserData
        result = append(result, userdata)

        packet := S_NEWUSERENTER{
            UserData: result,
        }

        broadCastAction := &BroadCastAction{
            Room:    r,
            Protocl: S_NewuserEnter,
            Packet:  packet,
        }

        r.pushAction(0, broadCastAction)
    }
}
```

GameRoom

```go
func (r *ChatRoom) LeaveUser(userUID string) {

    packet := &S_CHANNELLEAVE{
        Uuid: userUID,
    }

    removeAction := &ChannelUserRemoveAction{
        Room:       r,
        TargetUser: userUID,
    }

    broadCastAction := &BroadCastAction{
        Room:    r,
        Protocl: S_Channelleave,
        Packet:  packet,
    }

    r.pushAction(0, removeAction)
    r.pushAction(0, broadCastAction)
}

func (r *ChatRoom) pushAction(afterTick int, action IJob
    r.PendingList.Push(r.tick, afterTick, action)
}

func FindChannel(roomname string) (*ChatRoom, bool) {
    target, exist := rooms.ChannalRooms[roomname]

    if exist == false {
        return nil, exist
    }

    return target, exist
}
```

```go
func FindRoom(roomtype int, roomname string) (*ChatRo
    if roomtype == TYPE_MATCH {
        target, exist := rooms.MatchRooms[roomname]

        if exist == false {
            return nil, exist
        }

        return target, exist
    } else {
        target, exist := rooms.PrivateRooms[roomname]

        if exist == false {
            return nil, exist
        }

        return target, exist
    }
}

func Findmatch(roomname string) (*ChatRoom, bool) {
    target, exist := rooms.PrivateRooms[roomname]

    if exist == false {
        return nil, exist
    }

    return target, exist
}
```

```go
func (r *ChatRoom) GetEveryUsers() []data.UserData {
    var users []data.UserData

    currentUsermap := r.UserList

    for _, user := range currentUsermap {

        data := data.UserData{
            UserUid:  user.uid,
            Username: user.usernick,
        }

        users = append(users, data)
    }

    return users
}

func BroadCastChating(r *ChatRoom, userUid string, chat string) {
    targetUser := r.UserList[userUid]

    if targetUser == nil {
        return
    }

    userdata := data.UserData{
        UserUid:  targetUser.uid,
        Username: targetUser.usernick,
    }

    var result []data.UserData
    result = append(result, userdata)

    packet := &S_CHAT{
        UserData: result,
        Chat:     chat,
    }

    broadCastAction := &BroadCastAction{
        Room:    r,
        Protocl: S_chat,
        Packet:  packet,
    }

    databasepack.RecordChatMessage(r.RoomName, targetUser.uid, targetUser.usernick, chat)
    r.pushAction(0, broadCastAction)
}
```

GameRoom

```go
func BroadCastSystemChat(message string) {
    packet := S_SYSCHAT{
        Message: message,
    }

    // for _, val := range GetEverySession(Sessions) {
    //  WriteMessage(val, S_Syschat, packet)
    // }

    for _, val := range rooms.ChannalRooms {
        broadCastAction := &BroadCastAction{
            Room:    val,
            Protocl: S_Syschat,
            Packet:  packet,
        }

        val.pushAction(0, broadCastAction)
    }
}

func BootingChannel() {
    for i := 0; i < 1; i++ {
        Roomname := fmt.Sprintf("Channel_%d", i)
        rooms.startChatChannel(Roomname)
    }
}

func (m *Rooms) startChatChannel(name string) bool {
    m.m.Lock()
    defer m.m.Unlock()

    newRoom := &ChatRoom{
        UserList: make(map[string]*chatSession),

        RoomType: TYPE_CHANNEL,
        RoomName: name,
        StopRoom: false,
    }

    m.ChannalRooms[name] = newRoom
    go tickRoom(newRoom)
    return true
}
```

GameRoom

- startChatChannel에서 go 언어로 tickRoom이 병렬처리로 실행됩니다. 이는 위에서 JobTimerQueue의 내용을 전부 Flush하는 것입니다.
- EnterUser는 먼저 들어온 유저에게 방에 들어갔다는 알림을, 다른 유저에게는 해당유저가 들어왔다는 알림을 날리면서 서로의 정보에 대해 교환하고, Job으로 일이 처리되니 별다른 락을 걸지 않고, 계속해서 유사 스레드가 일을 하도록 처리해줍니다.
- 반대로 LeaveUser는 모든 유저에게 해당 유저가 나갔다고 알림을 보냅니다.

PacketHandler

```go
import (
    data "NurhymChat/Data"
    utils "NurhymChat/Utils"
    "encoding/json"
    "fmt"
)

var p_handler *PacketHandler

type PacketHandler struct {
    Handler map[int]func(p *chatSession, payload string)
}

func InitPacketHandler() {
    p_handler = &PacketHandler{
        Handler: make(map[int]func(p *chatSession, payload string)),
    }

    p_handler.Handler[C_ReqChannelList] = Handle_C_ChannelList
    p_handler.Handler[C_ChannelEnter] = Handle_C_ChannelEnter
    p_handler.Handler[C_chat] = Handle_C_Chat
    p_handler.Handler[C_Leavechannel] = Handle_C_Leave
    p_handler.Handler[C_RequestChatMatch] = Handle_C_RequestChatMatch
    p_handler.Handler[C_CancelMatch] = Handle_C_CancelMatch
    p_handler.Handler[C_Matchenter] = Handle_C_MatchEnter
    p_handler.Handler[C_Matchchat] = Handle_C_Matchchat
    p_handler.Handler[C_MatchLeave] = Handle_C_MatchLeave
    p_handler.Handler[C_MakeRoom] = Handle_C_MakeRoom
    p_handler.Handler[C_Roomlist] = Handle_C_RoomList
    p_handler.Handler[C_YieldHost] = Handle_C_YieldHost
}

func Handle_C_ChannelList(p *chatSession, payload string) {
    var res S_CHANNELLIST
    for _, val := range rooms.ChannalRooms {

        roomstatus := &data.ChannelList{
            ChannelName:   val.RoomName,
            CurrentPeople: len(val.UserList),
            MaxPeople:     MAX_PEOPLE,
        }

        res.ChannelList = append(res.ChannelList, *roomstatus)
    }

    WriteMessage(p, S_ChannelList, res)
}
```

PacketHandler

```
const (
    S_Test = iota + 1
    S_ReqTocke
    C_ResTocke
    S_ChannelList
    C_ReqChannelList
    C_ChannelEnter
    S_ChannelEnter
    S_NewuserEnter
    C_chat
    S_chat
    S_Syschat
    S_Kickfromchannel
    S_CloseChannel
    S_Channelleave
    C_Leavechannel
    C_RequestChatMatch
    C_CancelMatch
    S_Matchresult
    C_Matchenter
    S_Matchenter
    S_Matchnewuser
    C_Matchchat
    S_Matchchat
    C_MatchLeave
    S_MatchLeave
    C_MakeRoom
    C_YieldHost
    S_YieldHost
    C_KickUser
    S_KickUser
    C_Roomlist
    S_Roomlist
    S_RoomRemove
    S_ChatLog
    C_DirectChat
    S_DirectChat
)

type IPacket interface{}
```

```
type S_REQTOCKE struct {
    IPacket
}

type C_RESTOKEN struct {
    IPacket
    UserID string `json:"userid"`
    Token  string `json:"token"`
}

type S_CHANNELLIST struct {
    IPacket
    ChannelList []data.ChannelList `json:"channelLists"`
}

type C_REQCHANNELLIST struct {
    IPacket
}

type C_CHANNELENTER struct {
    IPacket
    ChannelName string `json:"channelname"`
}

type S_CHANNELENTER struct {
    IPacket
    JoinSuccess bool            `json:"joinsuccess"`
    Roomname    string          `json:"channelname"`
    UserData    []data.UserData `json:"userdatas"`
}

type S_NEWUSERENTER struct {
    IPacket
    UserData []data.UserData `json:"userDatas"`
}

type C_CHAT struct {
    IPacket
    Chat string `json:"chat"`
}
```

PacketHandler

- Session에서 m.Conn으로 패킷을 받아들이는 작업을 했었습니다. 이 작업으로 패킷은 직렬화가 가능하고, 이를 실행할 함수가 필요한데, packetHandler로 실행했습니다.
- 핸들러 맵에서는 map으로 세션과 string값을 가지고 있는 함수를 기억해두고, 함수가 conn이 패스가 되면 해당 함수로 들어와 패킷을 실행하게 됩니다.
- 실행함수는 따로 처리를 해줬어야 했습니다.