

게임 서버 프로그래밍

게임 설명서

2020182042 최준하

목차

table of contents

- 1 프로토콜
- 2 자료구조
- 3 알고리즘



Packet ID, etc

```
constexpr int PORT_NUM = 4000;
constexpr int BUF_SIZE = 200;
constexpr int NAME_SIZE = 20;
constexpr int CHAT_SIZE = 350;

constexpr int MAX_USER = 10000;
constexpr int MAX_NPC = 200000;
constexpr int MAX_BLOCK = 40000;

constexpr int W_WIDTH = 2000;
constexpr int W_HEIGHT = 2000;

constexpr int SECTOR_SIZE = 10;

// Packet ID
constexpr char CS_LOGIN = 0;
constexpr char CS_MOVE = 1;
constexpr char CS_CHAT = 2;
constexpr char CS_ATTACK = 3; // 4 방향 공격
constexpr char CS_TELEPORT = 4; // RANDOM한 위치로 텔레포트
constexpr char CS_LOGOUT = 5; // 클라이언트 종료

constexpr char SC_LOGIN_INFO = 2;
constexpr char SC_LOGIN_FAIL = 3;
constexpr char SC_ADD_OBJECT = 4;
constexpr char SC_REMOVE_OBJECT = 5;
constexpr char SC_MOVE_OBJECT = 6;
constexpr char SC_CHAT = 7;
constexpr char SC_STAT_CHANGE = 8;
constexpr char SC_ATTACK = 9; // 4 방향 공격

enum E_Visual { MARIO, SQUID, GUMBA, PLANT, BLOCK };
```

Client to Server

```
#pragma pack (push, 1)
struct CS_LOGIN_PACKET {
    unsigned short size;
    char type;
    char name[NAME_SIZE];
    char tester;
};

struct CS_MOVE_PACKET {
    unsigned short size;
    char type;
    char direction; // 0 : UP, 1 : DOWN, 2 : LEFT, 3 : RIGHT
    unsigned move_time;
};

struct CS_CHAT_PACKET {
    unsigned short size;
    char type; // 크기가 가변이다, mess가 작으면 size도 줄어든다.
    char mess[CHAT_SIZE];
};

struct CS_TELEPORT_PACKET {
    unsigned short size;
    char type; // 랜덤으로 텔레포트 하는 패킷, 동점 테스트에 필요
};

struct CS_ATTACK_PACKET {
    unsigned short size;
    char type;
};

struct CS_LOGOUT_PACKET {
    unsigned short size;
    char type;
};

#pragma pack (pop)
```

Server to Client

```
struct SC_LOGIN_INFO_PACKET {
    unsigned short size;
    char type;
    int visual; // 종족, 성별등을 구분할 때 사용
    int id;
    int hp;
    int max_hp;
    int exp;
    int level;
    short x, y;
};

struct SC_ADD_OBJECT_PACKET {
    unsigned short size;
    char type;
    int id;
    int visual; // 어떻게 생긴 OBJECT인가를 지시
    short x, y;
    char name[NAME_SIZE];
};

struct SC_REMOVE_OBJECT_PACKET {
    unsigned short size;
    char type;
    int id;
};

struct SC_MOVE_OBJECT_PACKET {
    unsigned short size;
    char type;
    int id;
    short x, y;
    unsigned int move_time;
};

struct SC_CHAT_PACKET {
    unsigned short size;
    char type;
    int id;
    char name[NAME_SIZE];
    char mess[CHAT_SIZE];
};

struct SC_LOGIN_FAIL_PACKET {
    unsigned short size;
    char type;
};

struct SC_STAT_CHANGE_PACKET {
    unsigned short size;
    char type;
    int hp;
    int max_hp;
    int exp;
    int level;
};

struct SC_ATTACK_PACKET {
    unsigned short size;
    char type;
    short x, y;
};

#pragma pack (pop)
```

SESSION

SESSION.h

```
enum S_STATE { ST_FREE, ST_ALLOC, ST_INGAME };
enum S_TYPE { AGRO, PEACE };
class SESSION {
    OVER_EXP _recv_over;
public:
    mutex _s_lock;
    S_STATE _state;
    atomic_bool _active;
    atomic_bool _npc;
    int _id;
    SOCKET _socket;
    short _x, _y;
    char _name[NAME_SIZE];
    chrono::system_clock::time_point _rm_time;
    unordered_set<SESSION*> view_list;
    mutex _vl_l;

    lua_State* _l;
    mutex _ll;
    atomic_bool _way_move;
    short _way;

    int _sector_x;
    int _sector_y;

    int _prev_remain;
    int _last_move_time;

    int hp;
    int max_hp;
    int exp;
    int level;
    int visual; // 종족, 성별등을 구분할 때 사용
    int atk;
    S_TYPE type;
};
```

```
bool invisible;
bool healing;

public:
    SESSION() { ... }

    ~SESSION() {}

    void set_sector();

    void do_recv();

    void do_send(void* packet);
    void send_login_fail_packet(const char* name);
    void send_login_info_packet();
    // ID 대신 SESSION을 넘겨보자
    void send_move_packet(SESSION& obj);
    void send_add_object_packet(SESSION& obj);
    void send_chat_packet(SESSION& obj, const char* mess, const char* name);
    void send_remove_object_packet(SESSION& obj);
    void send_attack_object_packet();
    void send_stat_change_packet();
    void do_chat(CS_CHAT_PACKET* p);
    void do_random_move();
    void do_way_move(SESSION& obj);
};
```

main.cpp

```
array<SESSION, MAX_NPC + MAX_USER + MAX_BLOCK> objects;
```

EVENT

EVENT.h

```
enum EVENT_TYPE { EV_RANDOM_MOVE, EV_HEAL, EV_ATTACK, EV_WAY_MOVE, EV_INVISIBLE };
class EVENT {
public:
    int obj_id;
    std::chrono::system_clock::time_point wakeup_time;
    EVENT_TYPE e_type;
    int target_id;
    bool operator< (const EVENT& rhs) const {
        return wakeup_time > rhs.wakeup_time;
    }
};
```

main.cpp

```
concurrency::concurrent_priority_queue<EVENT> g_event_queue;

void add_timer(int id, EVENT_TYPE type, int ms, int p_id)
{
    auto wakeup_time = std::chrono::system_clock::now() + std::chrono::milliseconds(ms);
    g_event_queue.push({ id, wakeup_time, type, p_id });
}
```

OVER_EXP

OVER_EXP.h

```
enum COMP_TYPE { OP_ACCEPT, OP_RECV, OP_SEND, OP_RANDOM_MOVE, OP_WAY_MOVE, OP_PLAYER_MOVE, OP_REGEN };

class OVER_EXP {
public:
    WSAOVERLAPPED _over;
    WSABUF _wsabuf;
    char _send_buf[BUF_SIZE];
    COMP_TYPE _comp_type;
    int _ai_target_obj;
    OVER_EXP()
    {
        _wsabuf.len = BUF_SIZE;
        _wsabuf.buf = _send_buf;
        _comp_type = OP_RECV;
        ZeroMemory(&_over, sizeof(_over));
    }
    OVER_EXP(char* packet)
    {
        _wsabuf.len = packet[0];
        _wsabuf.buf = _send_buf;
        ZeroMemory(&_over, sizeof(_over));
        _comp_type = OP_SEND;
        memcpy(_send_buf, packet, packet[0]);
    }
};
```

SESSION.h

```
class SESSION {
    OVER_EXP _recv_over;
```


DATA BASE

DataBase.h

```
struct Data {
    short x, y;
    int hp;
    int maxhp;
    int level;
    int exp;
};

class CDataBase
{
public:
    ~CDataBase();
    bool InitializeDB();
    void CloseDB();
    bool UpdateUserData(char* userId, Data data);
    bool FetchUserData();
    bool FindUserData(char* userId);
    Data GetUserData(char* userId);
    bool CreateUserData(char* userId);
private:
    SQLHENV henv = nullptr;
    SQLHDBC hdbc = nullptr;
    SQLHSTMT hstmt = nullptr;
    SQLRETURN retcode;
};
```

main.cpp

```
CDataBase db;
```

```
void db_update()
{
    while (true) {
        for (int i = USER_START; i < USER_START + MAX_USER; ++i) {
            if (objects[i].state != ST_INGAME) continue;
            db.UpdateUserData(objects[i].name, { objects[i].x, objects[i].y, objects[i].hp, objects[i].max_hp, objects[i].level, objects[i].exp });
        }
        this_thread::sleep_for(std::chrono::seconds(5));
    }
}
```

서버 패킷 재조립

```

case OP_RECV: {
    int remain_data = num_bytes + objects[key]._prev_remain;
    char* p = ex_over->_send_buf;
    while (remain_data > 0) {
        int packet_size = p[0];
        if (packet_size ≤ remain_data) {
            process_packet(static_cast<int>(key), p);
            p = p + packet_size;
            remain_data = remain_data - packet_size;
        }
        else break;
    }
    objects[key]._prev_remain = remain_data;
    if (remain_data > 0) {
        memcpy(ex_over->_send_buf, p, remain_data);
    }
    objects[key].do_rcv();
    break;
}

```

클라이언트 패킷 재조립

```

void process_data(char* net_buf, size_t io_byte)
{
    char* ptr = net_buf;
    static size_t in_packet_size = 0;
    static size_t saved_packet_size = 0;
    static char packet_buffer[BUF_SIZE];

    while (0 ≠ io_byte) {
        if (0 = in_packet_size) in_packet_size = ptr[0];
        if (io_byte + saved_packet_size ≥ in_packet_size) {
            memcpy(packet_buffer + saved_packet_size, ptr, in_packet_size - saved_packet_size);
            ProcessPacket(packet_buffer);
            ptr += in_packet_size - saved_packet_size;
            io_byte -= in_packet_size - saved_packet_size;
            in_packet_size = 0;
            saved_packet_size = 0;
        }
        else {
            memcpy(packet_buffer + saved_packet_size, ptr, io_byte);
            saved_packet_size += io_byte;
            io_byte = 0;
        }
    }
}

```

```
void worker_thread()
{
    while (true) {
        DWORD num_bytes;
        ULONG_PTR key;
        WSAOVERLAPPED* over = nullptr;
        BOOL ret = GetQueuedCompletionStatus(h_iocp, &num_bytes, &key, &over, INFINITE);
        OVER_EXP* ex_over = reinterpret_cast<OVER_EXP*>(over);
        if (FALSE == ret) { ... }

        if ((0 == num_bytes) && ((ex_over->_comp_type == OP_RECV) || (ex_over->_comp_type == OP_SEND))) { ... }

        switch (ex_over->_comp_type) {
            case OP_ACCEPT: { ... }
            case OP_RECV: { ... }
            case OP_SEND: { ... }
                break;
            case OP_RANDOM_MOVE: { ... }
                break;
            case OP_WAY_MOVE: { ... }
                break;
            case OP_PLAYER_MOVE: { ... }
                break;
            case OP_REGEN: { ... }
                break;
        }
    }
}
```

OP_ACCEPT – 연결

OP_RECV – 데이터 수신

OP_SEND – 데이터 송신

OP_RANDOM_MOVE – NPC 랜덤 이동

OP_WAY_MOVE – NPC 랜덤 이동

OP_PLAYER_MOVE –
플레이어와 겹쳤는지 체크(lua script)

OP_REGEN – NPC 부활

알고리즘

```
void do_timer()
{
    using namespace chrono;
    while (true) {
        if (!g_event_queue.empty()) {
            EVENT ev;
            if (g_event_queue.try_pop(ev)) {
                if (ev.wakeup_time < system_clock::now()) {
                    if (ev.e_type == EV_HEAL) { // 5초마다 체력 리젠
                        if (objects[ev.obj_id].hp + objects[ev.obj_id].max_hp / 10 < objects[ev.obj_id].max_hp) {
                            objects[ev.obj_id].hp += objects[ev.obj_id].max_hp / 10;
                            objects[ev.obj_id].send_stat_change_packet();
                            add_timer(ev.obj_id, EV_HEAL, 5000, 0);
                        }
                    }
                    else {
                        objects[ev.obj_id].hp = objects[ev.obj_id].max_hp;
                        objects[ev.obj_id].send_stat_change_packet();
                        objects[ev.obj_id].healing = false;
                    }
                }
                else if (ev.e_type == EV_INVISIBLE) { // 부활
                    objects[ev.obj_id].invisible = false;
                    OVER_EXP* ov = new OVER_EXP;
                    ov->_comp_type = OP_REGEN;
                    if (player_exist(ev.obj_id))
                        PostQueuedCompletionStatus(h_iocp, 1, ev.obj_id, &ov->_over);
                    else
                        objects[ev.obj_id]._active = false;
                }
            }
        }
    }
}
```

EV_RANDOM_MOVE – NPC 이동

EV_HEAL – 플레이어 힐

EV_WAY_MOVE – NPC 이동

EV_INVISIBLE – NPC 부활

알고리즘

CS_MOVE 패킷 처리 부분 중 일부분

```
for (int i = objects[c_id]._sector_x - 1; i ≤ objects[c_id]._sector_x + 1; ++i) {
    for (int j = objects[c_id]._sector_y - 1; j ≤ objects[c_id]._sector_y + 1; ++j)
        if (i ≥ 0 && i < W_WIDTH / SECTOR_SIZE && j ≥ 0 && j < W_HEIGHT / SECTOR_SIZE) {
            sec_l[i][j].lock();
            for (auto obj : sectors[i][j].objects)
                if (obj->_id ≠ objects[c_id]._id && can_see(&objects[c_id], obj) && !obj->invisible) {
                    new_viewlist.insert(obj);
                    if (is_npc(obj->_id) && obj->visual ≠ BLOCK && !obj->invisible) {
                        OVER_EXP* exover = new OVER_EXP;
                        exover->_comp_type = OP_PLAYER_MOVE;
                        exover->_ai_target_obj = objects[c_id]._id;
                        PostQueuedCompletionStatus(h_iocp, 1, obj->_id, &exover->_over);
                        if (obj->_active = false && obj->visual ≠ PLANT) {
                            bool f = false;
                            if (true = atomic_compare_exchange_strong(&obj->_active, &f, true)) {
                                if (obj->type = AGRO)
                                    add_timer(obj->_id, EV_WAY_MOVE, 1000, objects[c_id]._id);
                                else
                                    add_timer(obj->_id, EV_RANDOM_MOVE, 1000, objects[c_id]._id);
                            }
                        }
                    }
                }
            sec_l[i][j].unlock();
        }
    }
```

인근 섹터에 있는 오브젝트중에서
시야 내에 있으면 viewlist 추가

Npc이고 block이 아니고
살아있으면
충돌체크와 함께 timer에 추가