배틀로얄 생존 FPS

- The Last Soldier -

이종현 기술분석서

2019. 12.

1 클래스 구조
클래스 구조와 상속 관계

서버 모듈 2
C++ IOCP, 가변길이 패킷

3 기술
패턴, 동기화, DB, 최적화

클라이언트 4
C# TcpClient

클래스 구조

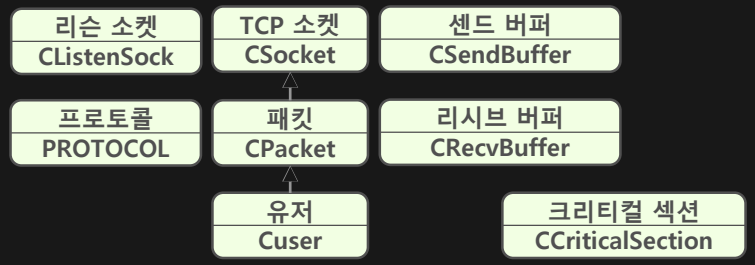
매니저 클래스
Manager Class

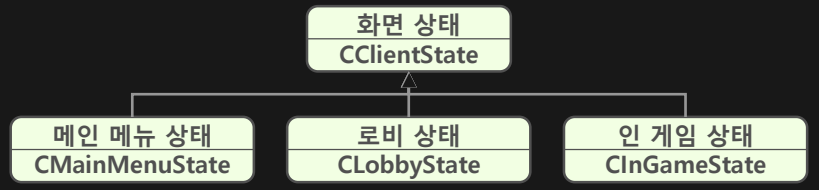
- 데이터 저장과 관리
- 역할에 맞는 기능을 제공
- 싱글턴 패턴 사용

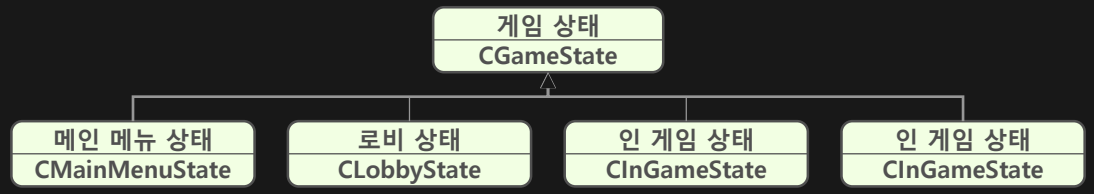
	로그 매니저 CLogManager	IOCP 매니저 CIOCPManager	DB 매니저 CDBManager	유저 매니저 CUserManager	아이템 인포 매니저 CItemInfoManager	아이템 매니저 CItemManager
메인 매니저 CMainManager	섹터 매니저 CSectorManager	룸 매니저 CRoomManager	매치 매니저 CMatchManager	씬 매니저 CSceneManager	메인 메뉴 씬 매니저 CMainMenuManager	로비 씬 매니저 CLobbyManager
	인 게임 씬 매니저 CInGameManager	컨트롤 매니저 CControlManager	플레이어 매니저 CPlayerManager	게임 맵 매니저 CGameMapManager	게임 매니저 CGameManager	로그인 매니저 CLoginManager

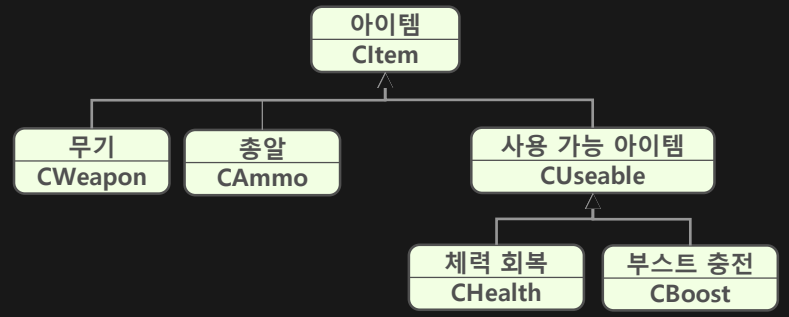
클래스
Class

- 캡슐화를 사용하여 정보 은닉
- 추상화와 상속을 사용하여 코드 재사용 및 유지 보수 용이









서버 모듈

IOCP 모델

I/O Completion Port Model

- 동시 접속한 많은 클라이언트들을 처리하기 위해 사용

- 추상화 클래스 CIOCPModel을 상속받아 설계

```cpp
/*
    CIOCPModel
        - IOCP 모델 서버의 기본 프레임 워크 입니다.

        #1. 모든 IOCP 서버에서 공통적으로 사용하는 최소한의 기능만 구현하는 추상화 클래스 입니다.


        !1. 이 클래스를 상속받은 클래스에서는 가상함수를 구현해서 사용해야 합니다.
*/
class CIOCPModel
{
protected:
    // 입출력 완료 포트
    HANDLE  m_hcp;
public:
    // 소켓의 Input Output 완료 통보를 기다리고 반환받은 구조체를 토대로 작업을 진행하는 스레드 입니다.
    static DWORD WINAPI WorkerThread(LPVOID arg);
public:
    bool          IOCPInit();
    bool          IOCPEnd();
    void          IOCPRun();

    HANDLE GetHcp() { return  m_hcp; }
public:
    // 새로 연결된 클라이언트 처리
    virtual bool Accept_IOCP(void*) = 0;
    // Recv(Read) 완료 처리
    virtual bool Recv_IOCP(void*,int) = 0;
    // Send(Write) 완료 처리
    virtual bool Send_IOCP(void*,int) = 0;
    // 연결 종료된 클라이언트 처리
    virtual bool Disconnect_IOCP(void*) = 0;
};
```

(사진) IOCPModel 클래스

프로토콜
Protocol

- 서버와 클라이언트 사이에서 패킷을 구분하기 위함

- 비트 연산을 이용해 하나에 패킷에 다수의 데이터를 추가하여 데이터 통신 횟수를 절약

```
class PROTOCOL
{
public:
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    /// 공통 적으로 사용할 프로토콜
    PROTOCOL_DATATYPE PT_REQ_MENUCHANGE = 1ll << 63;            // (클라 -> 서버) 메뉴(상태)를 변경 요청할때
    PROTOCOL_DATATYPE PT_RT_MENUCHANGE = 1ll << 62;             // (서버 -> 클라) 메뉴(상태) 변경 결과를 전송할때
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //// 메인 메뉴 상태 ////
    /// 메인 메뉴 상태의 큰 프로토콜
    PROTOCOL_DATATYPE PT_MAIN_LOGIN = 1ll << 49;                // 로그인 관련 프로토콜 일때

    /// 로그인 관련 프로토콜
    PROTOCOL_DATATYPE PT_MAIN_REQ_JOIN = 1ll << 0;              // (클라 -> 서버) 회원가입 메뉴에서 회원가입을 요청할때
    PROTOCOL_DATATYPE PT_MAIN_RT_JOIN_RESULT = 1ll << 1;        // (서버 -> 클라) 회원가입 후 결과를 전송할때
    PROTOCOL_DATATYPE PT_MAIN_REQ_LOGIN = 1ll << 2;             // (클라 -> 서버) 로그인 메뉴에서 로그인을 요청할때
    PROTOCOL_DATATYPE PT_MAIN_RT_LOGIN_RESULT = 1ll << 3;       // (서버 -> 클라) 로그인 후 결과를 전송할때
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    //// 로비 화면 상태 ////
    /// 로비 화면 상태의 큰 프로토콜
    PROTOCOL_DATATYPE PT_LB_MATCH = 1ll << 49;      // 매칭 관련 프로토콜 일때

    /// 매칭 관련 프로토콜
    PROTOCOL_DATATYPE PT_LB_MATCH_REQ_MATCH_START = 1ll << 0;           // (클라 -> 서버) 매칭 시작 요청을 보낼때
    PROTOCOL_DATATYPE PT_LB_MATCH_RT_MATCH_START = 1ll << 1;            // (서버 -> 클라) 매칭 시작 결과를 보내줄때
    PROTOCOL_DATATYPE PT_LB_MATCH_RT_MATCH_RESULT = 1ll << 2;           // (서버 -> 클라) 매칭 결과를 클라이언트에게 보내줄때 (매칭이 잡힐 경우)
    PROTOCOL_DATATYPE PT_LB_MATCH_REQ_MATCH_CANCEL = 1ll << 3;          // (클라 -> 서버) 매칭 중지 요청을 보낼때
    PROTOCOL_DATATYPE PT_LB_MATCH_RT_MATCH_CANCEL = 1ll << 4;           // (서버 -> 클라) 매칭 중지 결과를 보내줄때
    PROTOCOL_DATATYPE PT_LB_MATCH_RT_INSERT_ROOM = 1ll << 5;            // (서버 -> 클라) 매칭 대기중인 유저가 매칭이 잡혀 방에 접속할때
    ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

(사진) PROTOCOL 클래스

가변 길이 패킷 설계
Variable Length Packet Design

패킷 크기	Int32 (4 Byte)	패킷의 총 데이터 크기
프로토콜	Int64 (8 Byte)	패킷에 담긴 데이터의 정보를 구분하기 위해 사용하며 비트 연산 작업으로 패킷 하나에 여러 개의 정보를 담는다.
데이터 크기	Int32 (4 Byte)	다음 데이터의 총 크기를 저장한다.
데이터	데이터	실제 사용할 데이터

버퍼 클래스
Buffer Class

```cpp
/*
    CSendBuffer
        - 클라이언트로 보내는 패킷을 저장하는 버퍼 클래스 입니다.


        !1. 패킷을 전송할때 크기는 데이터를 저장하는 부분의 사이즈는 빠지기 때문에 GetSize() + sizeof(int)를 해주어야 합니다.
*/
class CSendBuffer
{
private:
    char* mDataBuffer;      // 패킷에서 프로토콜 + 데이터를 저장할 버퍼입니다.
    int mPacketSize;        // 패킷에 현재 저장된 프로토콜 + 데이터의 사이즈입니다.
public:
    // 생성자. 기본적으로 모든 패킷에는 프로토콜이 들어가므로 프로토콜을 생성자의 파라미터로 받는다.
    CSendBuffer(UINT64 _protocol);
    ~CSendBuffer();

    // 패킷에 프로토콜을 추가합니다.
    void AddProtocol(UINT64 _protocol);

    // 패킷에 데이터를 추가할때 호출하는 함수입니다.
    // 데이터의 시작 주소와 데이터 크기를 인자값으로 넘겨 받습니다.
    bool InsertData(void* _data, int _size);

    // 데이터를 추가할때 자주 사용하는 자료형을 미리 오버로딩 해줍니다.
    bool InsertData(int _data);
    bool InsertData(float _data);
    bool InsertData(UINT64 _data);



    char* GetBuffer()
    {
        return mDataBuffer;
    }
    int GetSize()
    {
        return mPacketSize;
    }
};
```

(사진) 송신용 버퍼 클래스

```cpp
/*
    CRecvBuffer
        - 클라이언트에게 받는 패킷을 저장하는 클래스 입니다.
*/
class CRecvBuffer
{
private:
    char* mDataBuffer;          // 수신한 패킷의 데이터를 저장하는 버퍼 입니다.
    int mPacketSize;            // 패킷의 전체 크기 입니다.
    int mImportedSize;          // 현재까지 가져온 데이터 크기 입니다.


public:
    // 수신 받은 데이터 배열과 크기를 생성자를 통해 넣어줍니다.
    CRecvBuffer(void* _data, int _size);
    ~CRecvBuffer();

    // 프로토콜을 읽어오기만 합니다. (읽어온 사이즈 증가X)
    bool GetProtocol(UINT64& _protocol);

    // 다음 데이터를 가져옵니다. (읽어온 사이즈 증가)
    bool GetNextData(void* _buf, int &size);
};
```

(사진) 수신용 버퍼 클래스

가변 길이 패킷 사용 예시
Variable Length Packet Usage Example

```cpp
// 플레이어의 현재 체력을 전송합니다.
bool CPlayerManager::Send_PlayerHP(CUser* _user)
{
    Sync synchro;

    // 유저의 플레이어 객체를 받아옵니다.
    CPlayer* getPlayer = _user->GetPlayer();

    // 송신용 버퍼를 생성합니다.
    CSendBuffer* sendBuffer = new CSendBuffer(PROTOCOL::PT_IG_PLAYERCONTROL | PROTOCOL::PT_IG_PLAYERCONTROL_RT_HP);

    // 멤버의 고유 멤버 번호를 패킷에 삽입합니다.
    sendBuffer->InsertData(_user->GetMemberCode());

    // 유저 플레이어의 현재 체력을 패킷에 삽입합니다.
    sendBuffer->InsertData(getPlayer->GetHP());

    // 유저의 팩패킷 함수를 호출하여 패킹 후 센드큐에 푸쉬 합니다.
    _user->PackPacket(sendBuffer->GetBuffer(), sendBuffer->GetSize());

    // 패킷의 내용을 전송합니다.
    _user->WSASendEX();

    // 사용한 송신 버퍼 객체를 제거합니다.
    delete sendBuffer;

    return true;
}
```

프로토콜 삽입
다수의 정보 삽입 가능

송신 버퍼에 데이터 삽입

패킷 전송

(사진) 송신 예제

기술

싱글턴 패턴
Singleton Pattern

- 하나의 프로그램에 단 하나의 객체만 생성

- 데이터 공유하기 용이

```cpp
class CDBManager : public CSynchronization<CDBManager>
{
public:
    // 이 클래스를 싱글턴 함수로 정의하기 위해 필요한 매크로를 사용합니다.
    DECLARE_SINGLE(CDBManager)
```

(사진) 클래스의 싱글턴 패턴 정의 예시

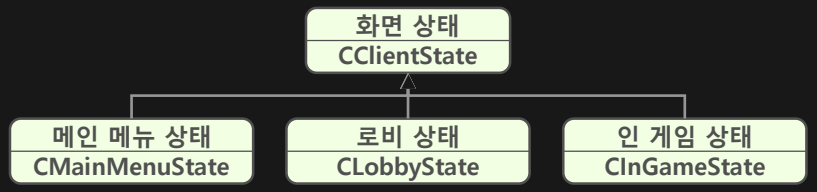
```cpp
/*
    DECLARE_SINGLE(Type)
        - 클래스를 싱글턴 객체로 선언하기 위한 전처리 매크로 입니다.

        #1. 클래스 내부에 DECLARE_SINGLE(클래스 명)을 선언하여 사용합니다.


        !1. 생성자 Type()와 소멸자 ~Type()의 정의를 구현해야 합니다.

        !2. 클래스 내부에 DECLARE_SINGLE를 선언하고 나면 필드는 private로 변경됩니다.
*/
#define DECLARE_SINGLE(Type)    \
    private:\
        static Type*    m_pInst;\
    public:\
        static Type* GetInst()\
        {\
            if(!m_pInst)\
                m_pInst = new Type;\
            return m_pInst;\
        }\
        static void DestroyInst()\
        {\
            if(m_pInst)\
                delete m_pInst;\
        }\
    private:\
        Type();\
        ~Type();

#define DEFINITION_SINGLE(Type) Type* Type::m_pInst = NULL;
#define GET_SINGLE(Type) Type::GetInst()
#define DESTROY_SINGLE(Type) Type::DestroyInst()
```

(사진) 싱글턴 패턴 정의 매크로

상태 패턴
State Pattern

- 상태에 따라 행동이 변하는 객체에 사용

- 상태마다 다른 행동을 하는 함수를 가상 함수로 구현



(그림) 상태 패턴 구조 예시

```cpp
/*
    CClientState
        - 클라이언트의 상태(씬)에 맞는 작업을 하는 상태패턴의 가장 부모가 되는 클래스 입니다.
*/
class CClientState : public CSynchronization<CClientState>
{
protected:
    // 이 스테이트 객체를 사용하는 유저
    CUser* mUser;

    // 상태를 변경해야 되는지 검사하는 변수
    bool mChangeState;
public:
    CClientState()
    {
        mChangeState = false;
    }
    CClientState(CUser* _user) : CClientState()
    {
        mUser = _user;
    }
    ~CClientState()
    {

    }
public:
    virtual bool Init() = 0;
    virtual bool Read() = 0;
    virtual bool Write() = 0;
};
```

(사진) 상태 패턴 인터페이스 예시

동기화
Synchronization

- 크리티컬 섹션을 이용한 동기화
- 템플릿을 이용해 클래스 마다 크리티컬 섹션 객체 공유
- 함수 단위로 임계 영역 설정
- 지역 변수로 메모리 할당 후 함수 종료 시 자동으로 소멸자 호출

```
/*
    CSynchronization
        - 템플릿 T 자료형의 크리티컬 섹션 객체를 생성합니다.

        #0. 자료형 마다 한개의 크리티컬 섹션 객체를 공유하여 사용합니다.

        #1. Sync 클래스의 생성자가 호출될때 임계영역을 잠그고 소멸자가 호출될때 임계영역에서 벗어납니다.

        #2. 사용하고자 하는 클래스에서 상속받고 클래스가 사용할 함수를 정의할때 지역 변수로 Sync 변수 선언 합니다.


        !0. 싱글턴과 같이 공통된 변수를 여러 스레드에서 접근할때만 사용해야 합니다.

        !1. 어트리뷰트 클래스의 경우에는 각자 독립된 크리티컬 섹션을 사용해야 합니다.
*/
template <typename T>
class CSynchronization
{
private:
    static CCriticalSection m_cs;
public:
    class Sync
    {
    public:
        Sync();
        ~Sync();
    };
};

template <typename T>
CCriticalSection CSynchronization<T>::m_cs;

template<typename T>
CSynchronization<T>::Sync::Sync()
{
    T::m_cs.Lock();
}

template<typename T>
CSynchronization<T>::Sync::~Sync()
{
    T::m_cs.Unlock();
}
```

```
/*
    CCriticalSection
        - 크리티컬 섹션 객체가 있는 객체입니다.
*/
class CCriticalSection
{
    CRITICAL_SECTION m_sec;
public:
    void Lock()
    {
        EnterCriticalSection(&m_sec);
    }

    void Unlock()
    {
        LeaveCriticalSection(&m_sec);
    }

    CCriticalSection()
    {
        InitializeCriticalSection(&m_sec);
    }

    ~CCriticalSection()
    {
        DeleteCriticalSection(&m_sec);
    }
};
```

(사진) 동기화 객체

동기화 사용 예시
Synchronization Usage Example

```cpp
/*
    CItemInfoManager
        - 데이터 베이스 서버에서 받아온 아이템의 기본 정보를 저정하고 관리하는 매니저 클래스 입니다.

        #1. 서버 실행시 데이터 베이스에서 아이템에 대한 정보를 모두 받아와 저장합니다.


        !1. CDBManager 클래스의 초기화보다 늦게 초기화가 실행돼야 합니다.
*/
class CItemInfoManager : public CSynchronization<CItemInfoManager>
{
    // 이 클래스를 싱글턴 함수로 정의하기 위해 필요한 매크로를 사용합니다.
    DECLARE_SINGLE(CItemInfoManager)
private:
    // DB에서 불러온 아이템의 기본 정보를 저장한다.
    vector<CDBTable_Item*> mItemInfos;

    // DB에서 아이템의 정보를 불러와 저장한다.
    bool LoadItemInfo();

    // 아이템 코드 생성
    UINT64 MakeItemCode();
public:
    bool Init();
    bool End();

    // 랜덤하게 아이템을 하나 생성하여 반환해준다.
    CItem* CreateNewRandItem();
};
```

(사진) 크리티컬 섹션 객체를 상속받는 매니저 예시

```cpp
bool CItemInfoManager::LoadItemInfo()
{
    // 함수를 임계 영역으로 설정
    Sync synchro;

    /// DB에서 아이템 정보들을 가져와 저장한다.

    // DB에서 데이터 가져와 데이터 테이블 구조체에 저장하여 받아오기 위한 변수
    CDBTable_Item getItemInfo;

    // 디비에서 아이템 정보를 읽어오는 함수를 호출한다.
    if (GET_SINGLE(CDBManager)->RunLoadItemInfo() == false)
    {
        return false;
    }

    // 아이템 정보를 가져오는 함수를 호출하고 데이터가 있다면 반복문을 실행한다.
    while (GET_SINGLE(CDBManager)->SearchGetData(getItemInfo))
    {
        // 아이템 정보 테이블을 리스트에 저장한다.
        mItemInfos.push_back(new CDBTable_Item(getItemInfo));
    }

    // 검색 마무리 함수 호출
    GET_SINGLE(CDBManager)->SearchEnd();


    return true;
}
```

(사진) 크리티컬 섹션 객체 사용 예시

데이터 베이스
Data Base

- MySQL 서버

DB 테이블
회원 정보
아이템 정보
게임 맵 정보
플레이어 스폰 구역 정보
아이템 스폰 구역 정보

(표) DB 테이블 종류

```
/*
    CDBManager
        - MySql 데이터 베이스 서버와 연결을 하고 쿼리문을 실행하고 결과를 처리하는 작업을 하는 매니저 클래스 입니다.


    #1. Run_*** 함수는 쿼리문을 실행하는 함수입니다.

    #2. 쿼리문 결과로 받은 테이블의 정보(MYSQL_RES* 데이터)를 직접 out 해주지 않고 DBRecordBuffer.h에 저장된 구조체를 이용하여 out 해줍니다


    !1. 데이터를 외부에서 반환 받을때는 Run함수 실행 -> (Loop)SearchGetData -> End의 로직을 지켜야 합니다.
*/
class CDBManager : public CSynchronization<CDBManager>
{
public:
    // 이 클래스를 싱글턴 함수로 정의하기 위해 필요한 매크로를 사용합니다.
    DECLARE_SINGLE(CDBManager)
private:
    MYSQL       m_MySQL;
    char        m_QueryBuffer[QUERY_BUFSIZE];   // SQL문을 실행할때 사용할 버퍼 문자열


    // 접근한 스레드가 쿼리문을 실행하여 가져오고 있는 데이터를 저장하는 맵
    map<DWORD, MYSQL_RES*> mSearchIters;
public:
    bool         Init();
    bool         End();

    ////// 쿼리문을 실행하고 그 결과를 다 받아온 뒤 마무리 작업을 하는 함수
    bool SearchEnd();
```

(사진) 데이터 베이스 매니저

최적화 - 섹터
Optimization - Sector

❖ 목적

- 게임이 진행되는 맵의 구역을 나누고 클라이언트에게 최소한의 데이터만 전송하여 불필요한 패킷 전송을 최소화

(사진) 섹터 구분 예시

- 구역을 나누는 최소 범위 (1 섹터)
- 기준이 되는 플레이어가 포함된 섹터
- 플레이어가 포함된 섹터 외에 데이터 통신 범위에 포함되는 구역

클라이언트

C# TcpClient

C# TcpClient

- Socket 클래스를 기본으로 사용하는 TcpClient 객체 사용
- 연결된 서버마다 수신용 Thread를 생성
- 유니티의 Update 호출 시 수신 받은 패킷 처리

```
/// <summary>
/// <para>서버와 연결을 담당하는 함수 입니다.</para>
/// </summary>
bool ConnectServer(NetworkServer _netIndex, String _ip, int _port)
{
    // 연결할 서버의 정보를 저장하는 객체를 생성합니다.
    CServer newServer = new CServer(_ip, _port);

    // 객체가 제대로 생성되지 않았다면 false를 반환합니다.
    if (newServer == null)
        return false;

    try
    {
        // TcpClient를 통해 서버와 연결을 시도합니다.
        newServer.Tc.Connect(IPAddress.Parse(_ip), _port);

        // TcpClient의 NetworkStream을 얻어옵니다.
        newServer.Ns = newServer.Tc.GetStream();

        // 서버에서 데이터를 수신받는 함수 ThreadRecv를 호출하는 스레드 생성합니다.
        newServer.RecvThread = new Thread(new ParameterizedThreadStart(ThreadRecv));

        // 스레드를 시작합니다. 인자값으로 생성한 CServer 객체를 넘겨줍니다.
        newServer.RecvThread.Start(newServer);


        // 서버 리스트에 연결에 성공한 서버 정보를 추가 해줍니다.
        serverList.Add(_netIndex, newServer);

        return true;
    }
    catch (Exception e)
    {
        Debug.Log(e.ToString());

        return false;
    }
}
```

(사진) TcpClient 서버 연결

서버 통신 과정
Server Communication Process