Mobile Network Framework Guide Line

[MNF Features]

Version (1.1.0)

- MNF code all refactoring
- Add Log [Server & Client] with LogWriter
- Add Mysql process

Version (1.0.1)

- TCP
- Client / Server
- Complete C# 2.0 native source code
- Stand alone library (DLL)
- Multi-Thread
- Event Driven
- Support [Binary/Json] message data

Tested Platforms

- Android
- Windows(x64)
- OSX (exclude LogServer)

[Binary/Json MessageDefine Class]

-> All message categories(enum) must be defined in MessageDefine class.

-> All messages must be defined in enum value.

-> All messages must be defined in MessageDefine class.

-> All messages must have "PACK_" prefix.

[Binary Message Define]

```
public partial class BinaryMessageDefine
{
    // ENUM
    public enum ENUM_CS_
    {
        CS_ECHO,
        ....
    }
    public enum ENUM_SC_
    {
        SC_ECHO,
        ....
    }

    [StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Ansi)]
    public class PACK_CS_ECHO
    {
        ....
    }

    [StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Ansi)]
    public class PACK_SC_ECHO
    {
        ....
    }
}
```

[Json Message Define]

```
public partial class JsonMessageDefine
{
    // ENUM
    public enum ENUM_ECHO_
    {
        ECHO,
        ....
    }

    [System.Serializable]
    public class PACK_ECHO : IJsonMessageHeader
    {
        ....
    }
}
```

[Binary Message Format]

-> All binary messages must have **[StructLayout]** attribute.

-> All binary messages must have the "BinaryMessageHeader" Field.
And sets messageSize and messageID.

-> If target program is C++ program, don't use class member variable in binary message.
Ex) struct POINT is correct, class POINT isn't correct.

-> If target program is C# program, you can use class / struct.

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct POINT
{
    [MarshalAs(UnmanagedType.U4)]
    public int x;
    [MarshalAs(UnmanagedType.U4)]
    public int y;
}

[StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Ansi)]
public class PACK_CS_ECHO
{
    public BinaryMessageHeader header_;

    [MarshalAs(UnmanagedType.Bool)]
    public bool boolField;

    [MarshalAs(UnmanagedType.I4)]
    public int intField;

    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 10)]
    public int[] intArrayField;

    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 100)]
    public string stringField;

    public POINT structField;

    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 10)]
    public POINT[] structArrayField;

    public PACK_CS_ECHO()
    {
        header_ = new BinaryMessageHeader();
        header_.messageSize = (short)MarshalRef.getManagedDataSize(this);
        header_.messageID = (ushort)ENUM_ECHO_.CS_ECHO;

        boolField = 0;
        intField = 0;
        intArrayField = new int[10];
        stringField = "";
        structField = new POINT();
        structArrayField = new POINT[10];
        for (int i = 0; i < structArrayField.Length; ++i)
            structArrayField[i] = new POINT();
    }
}
```

[Json Message Format]

-> All json messages must have **[System.Serializable]** attribute.

-> All json messages must be inherited from "IJsonMessageHeader".
"IJsonMessageHeader" is interface for Json message.

-> All json messages must implement "getMessageID()" function.
And must return messageID.

```
[System.Serializable]
public class Sandwich
{
    public string name;
    public string bread;
    public float price;
    public List<string> ingredients = new List<string>();

    public Sandwich()
    {
    }
}

[System.Serializable]
public class PACK_ECHO : IJsonMessageHeader
{
    public List<Sandwich> sandwiches = new List<Sandwich>();

    short IJsonMessageHeader.getMessageID()
    {
        return (short)ENUM_ECHO_.ECHO;
    }
}
```

[Message Enum with Binary/Json Message and Dispatcher]

-> All message enums must have correspond to [Binary/Json] message and dispatch function.
-> Prefix "On" to a message dispatcher function name.

<< Client / Server Message Define >>

```
public partial class BinaryMessageDefine
{
    // ENUM
    public enum ENUM_CS_          // CS, Client send > Server recv
    {
        CS_ECHO,
    }
    public enum ENUM_SC_          // SC, Server send > Client recv
    {
        SC_ECHO,
    }

    [StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Ansi)]
    public class PACK_CS_ECHO
    {
    }

    [StructLayout(LayoutKind.Sequential, Pack = 1, CharSet = CharSet.Ansi)]
    public class PACK_SC_ECHO
    {
    }
}
```

<< Server Code >>

```
public class BinaryMessageDispatcher : DispatchExporter
{
    public override bool onInit()
    {
        if (exportFunctionFromEnum<BinaryMessageDefine.ENUM_CS_>(this) == false)
          return false;

        if (exportClassFromEnum<BinaryMessageDefine, BinaryMessageDefine.ENUM_CS_>() == false)
          return false;

        return true;
    }

    private int onCS_ECHO(NF.Session session, object message)
    {
        var csEcho = (BinaryMessageDefine.PACK_CS_ECHO)packet;
        Console.WriteLine("Dispatch {0} Packet", csEcho.GetType().Name);

        var scEcho = new BinaryMessageDefine.PACK_SC_ECHO();
        session.send(scEcho);

        return 0;
    }
}
```

[System Message Dispatcher]

-> onAccept() : This function is called when client accepted MNF server.
-> onConnectSuccess() : This function is called when MNF client connected to server.
-> onConnectFail() : This function is called when MNF client didn't connect to server.
-> onDisconnect() : This function is called when session was disconnected from client / server.

```
public class SystemMessageCollection : SystemMessageDispatcher
{
    public override bool onInit()
    {
        return true;
    }

    public override int onAccept(Session session, object packet)
    {
        return 0;
    }

    public override int onConnectSuccess(Session session, object packet)
    {
        return 0;
    }

    public override int onConnectFail(Session session, object packet)
    {
        return 0;
    }

    public override int onDisconnect(Session session, object packet)
    {
        return 0;
    }
}
```

[Acceptor Class for Server]

-> Acceptor provides with session to accept.
-> Acceptor has two member function.
    1. create()
    2. start()
-> create() needs six arguments.
    1. server ip
    2. server port
    3. accept back log
    4. session type
    5. dispatch exporter type
    6. message formatter type
-> start() starts to accept client session.

It's Acceptor.create() example.

```
if (Acceptor.create(
    ServerIP
    , ServerPort
    , 500
    , typeof(ClientSession)
    , typeof(BinaryMessageDispatcher)
    , typeof(BinaryMessageFormatter)) == false)
{
    LogManager.Instance.Write("acceptor create failed");
    return false;
}
break;
```

Next, Acceptor.start() example.

```
if (Acceptor.start() == false)
{
    LogManager.Instance.Write("acceptor accept failed");
    return false;
}
```

[ConnectHelper Class for Client]

-> ConnectHelper provides with sesson to connect.
-> ConnectHelper has two member function.
   1. syncConnect()
   2. asyncConnect()
-> ConnectHelper.syncConnect() : supports synchronous connect.
   >> Don't notify to SystemMessageCollection.
-> ConnectHelpler.asyncConnect() : supports asynchronous connect.

It's ConnectHelper.asyncConnect() example.

```
public bool connectToServer()
{
    Session = ConnectHelper.asyncConnect(
        typeof(ServerSession)
        , typeof(TestNetworkMessageDispatcher)
        , typeof(BinaryMessageFormatter)
        , ServerIP
        , ServerPort) as ServerSession;

    return (Session != null);
}
```

And it's ConnectHelper.syncConnect() example.

```
LogSession = ConnectHelper.syncConnect(
    typeof(ServerSession)
    , typeof(TestNetworkMessageDispatcher)
    , typeof(JsonMessageFormatter)
    , ip
    , port) as ServerSession;

if (LogSession == null)
{
    LogManager.Instance.WriteError("async connect failed");
    return false;
}
```

[MNF_Helper Class]

-> MNF_Helper is very important singleton class object.
-> MNF_Helper has seven member functions.

1. run() : run MNF.
   >> It needs two arguments that **isRunThread** and **SystemMessageDispatcherType**.
2. stop() : stop MNF.
3. registDBMsgDispatcher() : regist DB Message Dispatcher.
4. registInterMsgDispatcher() : regist Inter Message Dispatcher.
5. pushDBMessage() : push DB Message to DB Message Dispatcher.
6. pushInterMessage() : push Inter Message to Inter Message Dispatcher.
7. dispatch_Network_Inter_Message() : dispatch network/inter message.

It's MNF_Helper.run() example.

```
int logType = (int)ENUM_LOG_TYPE.LOG_TYPE_ERROR;
logType |= (int)ENUM_LOG_TYPE.LOG_TYPE_NORMAL;
logType |= (int)ENUM_LOG_TYPE.LOG_TYPE_SYSTEM;
LogManager.Instance.SetLogWriter(new CLogWriter(ServerIP, LogServerPort), logType);

try
{
    IsInit = MNF_Helper.Instance.run(false, typeof(LocalSystemMessageCollection));
}
catch (Exception e)
{
    LogManager.Instance.Write(e.ToString());
    IsInit = false;
}

if (LogManager.Instance.init() == false)
    LogManager.Instance.Write("LogWriter init failed");
```

Important Note : Network/Inter messages are processed same thread.

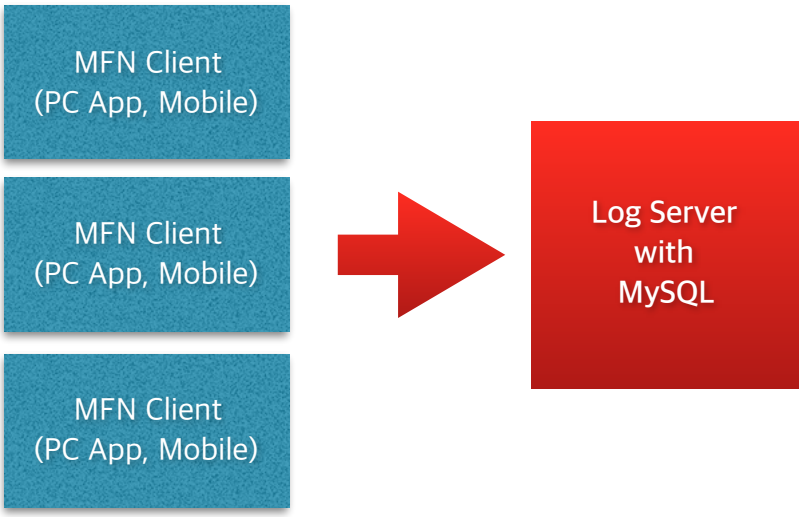
If isRunThread is true, don't call MNF_Helper.dispatch_Network_Inter_Message().
because MNF_Helper.dispatch_Network_Inter_Message() called another thread.

If isRunThread is false, call MNF_Helper.dispatch_Network_Inter_Message().

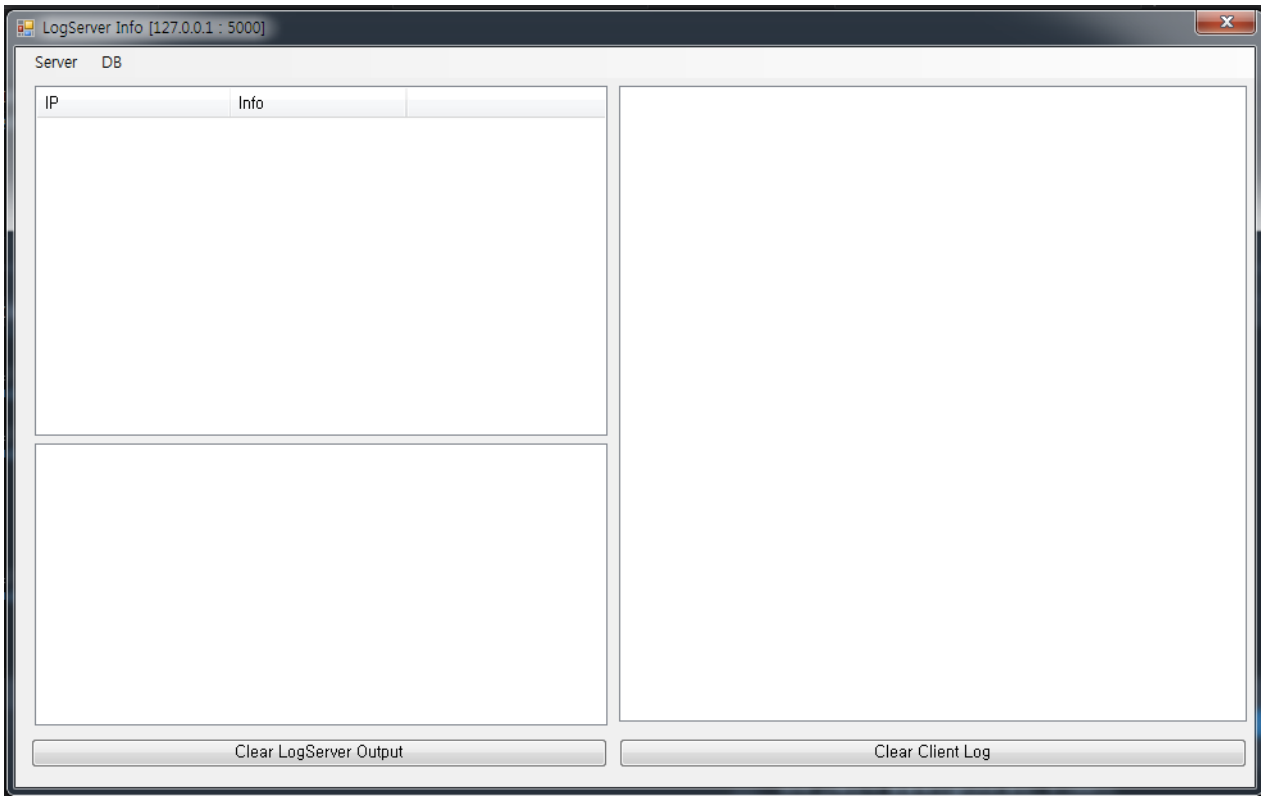
```
public void Update()
{
    if (IsInit == false)
        return;

    MNF_Helper.Instance.dipatch_Network_Inter_Message();
}
```

[Log Server & MySQL]

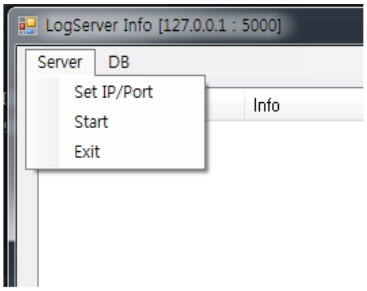
-> Log Server collects client logs, and push DB.



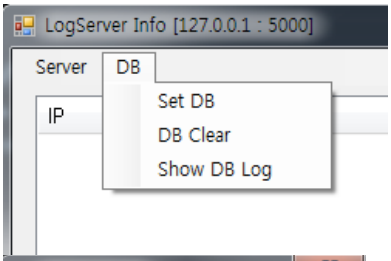
It's Log Server default view and Log Server have two menus
-> Server : Setting Server Information.
-> DB : Setting DB Information.

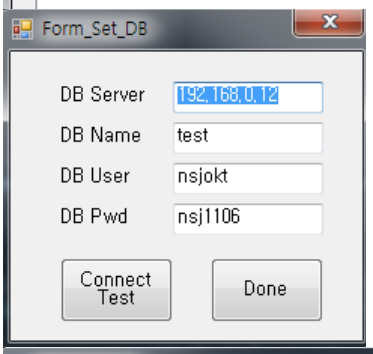


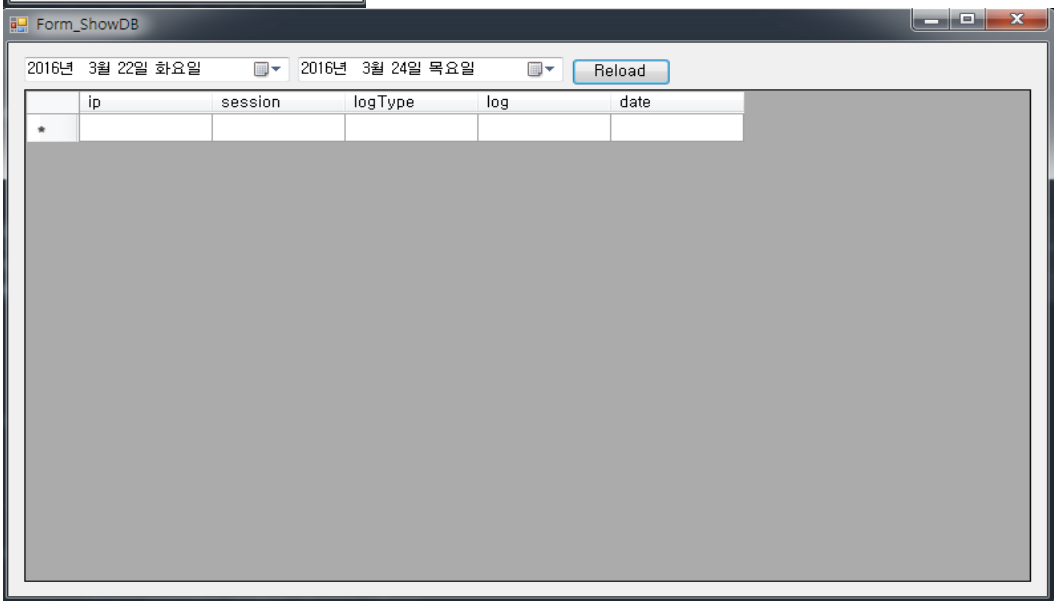
[Server Menu]
-> Set IP/Port : Input Server IP, Port.
-> Start : Start Log Server.



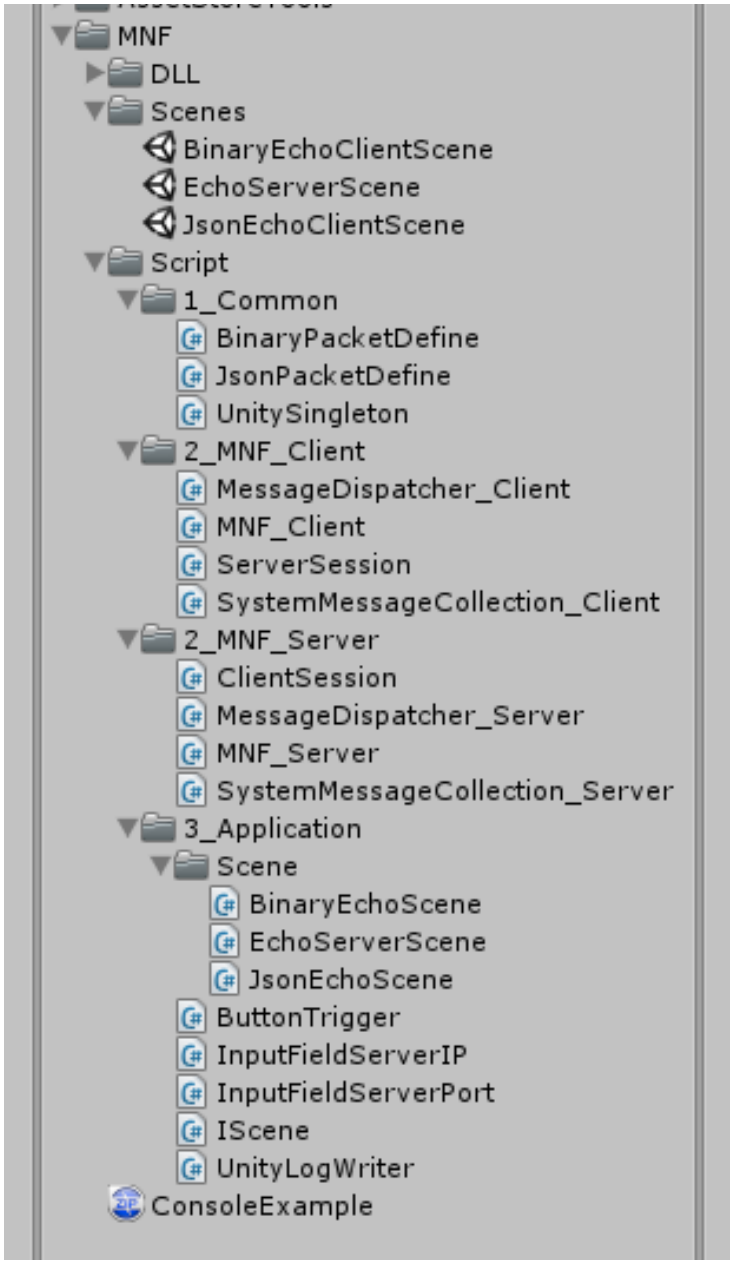
[DB Menu]
-> Set DB : Input DB Information.
-> DB Clear : Clear client log in DB.
-> Show DB Log : Show client log from DB.





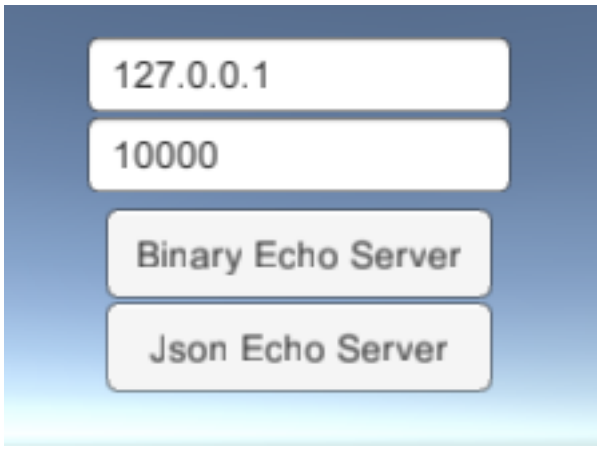


[Unity3d Scene]



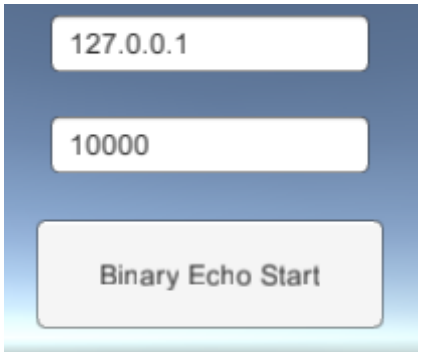
1. Echo Server Scene

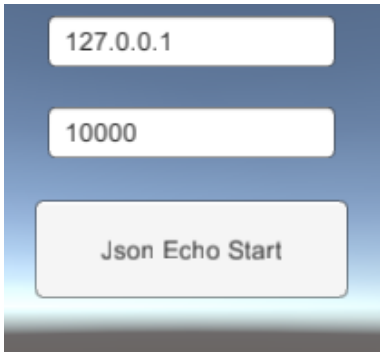
-> "127.0.0.1" is server IP.
-> 10000 is server port.
-> **"Binary Echo Server"** is button to start Binary Echo Server.
-> **"Json Echo Server"** is button to start Json Echo Server.



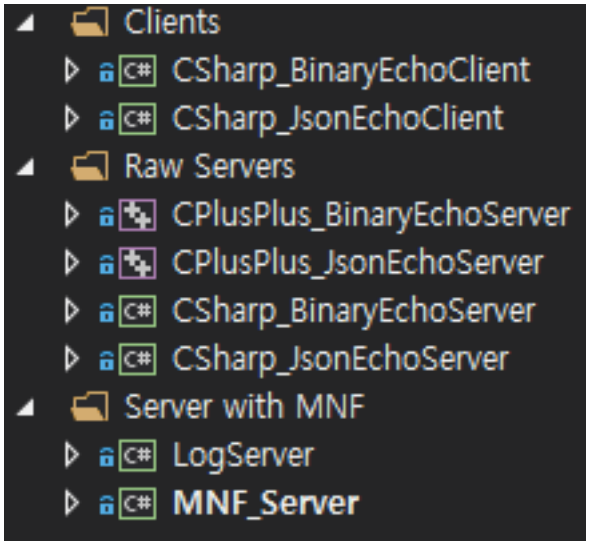
2. Binary Echo Client Scene

-> "127.0.0.1" is target server IP.
-> 10000 is target server port.
-> **"Binary Echo Start"** and **"Json Echo Start"** are button to connect to binary echo server.





Console Example



1. Clients

-> CSharp_BinaryEchoClient and CSharp_JsonEchoClient are MNF Echo Client.

-> Clients connect to Servers and MNF_Server.

-> LogWriter includes LogClient feature with MNF.Session.

2. Raw Servers

-> CPlusPlus_BinaryEchoServer and CPlusPlus_JsonEchoServer are native C++ program.

-> CSharp_BinaryEchoServer and CSharp_JsonEchoServer are native C# program.

3. Server with MNF

-> MNF_Server and LogServer are native C# programs.
-> MNF_Server and LogServer are developed by MNF_Library.

Exception

These are MNF exception string.

"IP EndPoint is invalid"
>> Server IP is invalid.

"Connect EndPoint is invalid"
>> Target server IP is invalid.

"Send data serialize failed"
>> [Binary / Json] message is invalid.

"Push size is zero"
"Readable buffer is zero"
"Request invalid readSize"
"Request popSize is zero"
"Pop buffer is empty"
"Request popSize invalid"
"CircularBuffer read failed"
>> Exception occurs in circular buffer .

"Circular buffer is full"
>> [Binary / Json] message size is over circular buffer.
>> Circular buffer size is 4096 bytes.

"Dispatcher found failed"
>> [Binary / Json] message dispatcher is not exist.