服务器引擎接口文档

1 接口列表

- NetServer 服务器基类，派生类实现网络事件响应方法，直接做业务处理
- NetHost连接的主机，用于该主机连接上的 recv\send\close 操作
- STNetServer 单线程版服务器基类，派生类实现网络事件响应方法，直接做业务处理
- STNetHost 单线程版连接的主机，用于该主机连接上的 recv\send\close 操作

2 NetServer

2.1 类说明

服务器基类
用户派生，实现自己的业务逻辑，在派生类构造函数中做初始化

2.2 服务器事件响应回调方法

- virtual void* Main(void* pParam)

  服务器启动主业务处理回调方法

  服务器业务线程不做任何事情，直接调用此方法，此方法退出，则服务器业务线程退出

  ※此线程退出，不表示服务器停止，这只是业务线程逻辑，服务器完全可以没有长期运行于后台的业务逻辑，只处理网络消息

  触发时机：服务器启动

  退出时机：
  Stop()被调用后，3s内不自己退出则被强制杀死
  IF业务中存在循环，可以使用IsOK()检查是否有Stop()被调用
  IF业务中存在线程挂起函数，需要在Stop()调用前自行发送信号唤醒线程正常结束

  用户也可以忽略此方法，自己创建管理业务线程

- bool IsOk()

  服务器状态检查，仅仅为main()方法中作为循环退出条件使用
  服务器Start()后返回true，Stop()后返回false

- virtual void OnConnect(NetHost &host)

  virtual void OnConnect(NetHost* pClient)v1.10版本原型

有新连接进来，业务处理回调方法
参数：
host连接进来的主机
用于数据io和一些其它主机操作，具体参考NetHost类

- virtual void OnCloseConnect(NetHost &host)
virtual void OnCloseConnect(NetHost* pClient) v1.10版本原型
有连接断开，业务处理回调方法
参数：
host连接断开的主机
用于调用ID()方法，标识断开对象，不必Close()，引擎已经Close()过了，
其它主机操作，具体参考NetHost类

- virtual void OnMsg(NetHost &host)
virtual void OnMsg(NetHost* pClient) v1.10版本原型
有数据可读，业务处理回调方法
参数：
host有数据可读的主机
用于调用ID()方法，标识断开对象，不必Close()，引擎已经Close()过了，
其它主机操作，具体参考NetHost类

2.3 服务器方法

- const char* Start()
运行服务器
成功返回NULL
失败返回失败原因

- void Stop()
关闭服务器

- void WaitStop()
等待服务器停止

- void SetAverageConnectCount(int count)
设置单个服务器进程可能承载的平均连接数，默认5000

- void SetReconnectTime(int nSecond);
设置自动重连时间,最小10s，不设置则，或设置小于等于0，服务器不重连
使用Connect()方法连接的地址断开时，系统会定时尝试重新连接
- void SetHeartTime(int nSecond)
设置心跳时间，不设置则，服务器不检查心跳

- void SetIOThreadCount(int nCount)

设置网络IO线程数量，建设设置为CPU数量的1~2倍

- void SetWorkThreadCount(int nCount)
  设置工作线程数（即OnConnect OnMsg OnClose的并发数）

- bool Listen(int port)
  监听某个端口，可多次调用监听多个端口

- bool Connect(const char *ip, int port)
  连接其它服务，可多次调用连接多个服务
  ※不要连接自身，引擎未做此测试，可能出现bug
- void BroadcastMsg(int *recvGroupIDs, int recvCount, char *msg, int msgsize, int *filterGroupIDs, int filterCount)
  广播消息
  向属于recvGroupIDs中任意一组，同时过滤掉属于filterGroupIDs中任意一组的主机，发送消息
  配合NetHost::InGroup(),NetHost::OutGroup()使用

  参数：

recvGroupIDs	要接收该消息的分组列表
recvCount	recvGroupIDs中分组数
msg	消息
msgsize	消息长度
filterGroupIDs	不能接收该消息的分组列表
filterCount	filterGroupIDs中分组数

  应用场景：
  游戏有很多地图，地图有唯一ID，作为分组ID
  NetHost1->InGroup(地图ID1) 玩家进入地图1
  NetHost2->InGroup(地图ID2) 玩家进入地图2
  BroadcastMsg({地图ID1, 地图ID2,}, 2,…)向地图1与地图2中的所有玩家发送消息

  例如：
  A B C3个主机
  A属于 1 2 4
  B属于 2 3 5
  C属于 1 3 5
  D属于 2 3
  E属于 3 5
  BroadcastMsg({1,3}, 2, msg, len, {5}, 1);
  向属于分组1或属于分组3,同时不属于分组5的主机发送消息，则AD都会收到消息，BCE被过滤

用户也可以不理会该方法，自己创建管理分组

- void SendMsg(int hostID, char *msg, int msgsize);
向某主机发送消息

参数：
hosteID 接收方id
msg 消息
msgsize 消息长度

与NetHost::Send()的区别
SendMsg()内部先在连接列表中，加锁查找对象，然后还是调用NetHost::Send()发送消息
在已经得到NetHost对象的情况下，直接NetHost::Send()效率最高，且不存在锁竞争。

- void CloseConnect(int hostID)
关闭与主机的连接

3 NetHost 接口

3.1 类说明

网络主机类
表示一个连接上来的主机，私有构造函数，只能由引擎创建，用户使用
该类是一个共享对象类，类似智能指针，所有通过复制产生的NetHost对象共享一个NetConnect指针。
所以不要复制地址与引用，因为引用引用计数不会改变，造成可能访问一个已经被释放的NetConnect指针
错误范例：
NetHost *pHost = &host;//复制地址，引用计数不会改变
vector<NetHost*> hostList;
hostList.push_back(&host);//复制地址，引用计数不会改变
正确范例：
NetHost safeHost = host;//复制对象，引用计数会增加1，在safeHost析构之前，NetConnect指针指向的内存绝对不会被释放
vector<NetHost> hostList;
hostList.push_back(host);//复制对象，引用计数会增加1，在对象从hostList中删除之前，NetConnect指针指向的内存绝对不会被释放

3.2 构造/析构函数

- Private NetHost(bool bIsServer)当前版本(V1.20)中已经删除

私有构造，用户不能自己创建该类对象，由引擎创建

- NetHost()
  默认构造函数

- NetHost(const NetHost& obj)
  复制构造函数
  为了NetConnect指针，引用计数+1

- virtual ~NetHost()
  析构函数
  为了NetConnect指针，引用计数-1，并在引用计数为0时，释放共享的NetConnect指针

3.3 方法

- void GetAddress(std::string &ip, int &port) V1.21新增
  主机地址
  如果你在NetServer希望得到对方地址，应该调用本方法，而不是GetServerAddress
  因为NetHost表示的就是对方主机，所以NetHost的主机地址就是对方地址

- void GetServerAddress(std::string &ip, int &port) V1.21新增
  服务器地址
  如果你希望知道对方主机连接到自己的哪个端口，应该调本方法，而不是GetAddress，因为GetAddress表示的是对方

- NetHost& operator=(const NetHost& obj);
  复制运算符重载，为了NetConnect指针，引用计数+1

- int ID()
  主机唯一标识
  ※实际就是与该主机连接的SOCKET句柄，但不可直接使用socket相关api来操作该socket的io与close。
  因为close时，底层需要做清理工作，如果直接使用socketclose()，则底层可能没机会执行清理工作,造成连接不可用
  io操作底层已经使用io缓冲管理，直接使用api io将跳过io缓冲管理，且会与底层io并发，将导致数据错乱

- uint32 GetLength()
  V1.52版中删除，用户不需要知道当前缓冲中数据长度，知道了也没用，因为该方法不会修改缓冲状态，用户就算知道数据长度不够，也不能直接从onmsg()里return，因为状态没变，return了还会循环进入onmsg，一直占用工作线程。应该只调用Recv操作缓冲，长度不够Recv会返回失败，同时会设置缓冲状态

取得已经到达数据长度

- bool Recv(unsigned char* pMsg, unsigned short uLength, bool bClearCache = true)

从接收缓冲中读数据

参数：

pMsg　　消息保存接收的数据

uLength　　想要接收的长度

bClearCache　是否将接收到的数据从缓存中清除，true删除，false保留

比如报文格式：2byte内容长度+报文内容

OnMsg()内接收逻辑如下

1.Recv(msg, 2, true);

2.解析msg得到内容长度，假设为256

3.Recv(msg, 256, true)

…

如果3.这里返回false，表示实际到达数据<256，不够读取

这时，用户有2种选择

选择1：

循环执行Recv 直到成功，如果不sleep，CPU直接100%，如果sleep，响应效率降低

选择2：

将256保存下来，直接return退出OnMsg，下次OnMsg触发时，再尝试Recv

优点，没有sleep,不吃CPU

缺点：用户代码难以组织，用户需要为所有连接维护一个int变量保存接收长度，也就是用户需要自己维护一个列表，在连接断开时，要从列表删除，工作繁杂

传递false到bClearCache，解决上面的问题

1.Recv(msg, 2, false);

2.解析msg得到内容长度，假设为256

3.Recv(msg, 256+2, true)//整个报文长度是256+2

如果Recv成功，直接处理

如果Recv失败，表示到达数据不够，因为1.哪里传递了false，报文长度信息不会从接收缓冲中删除，所以，用户可以直接return退出OnMsg,下次OnMsg触发时，还可以从连接上读到报文内容长度信息

返回值：

数据不够，直接返回false

无需阻塞模式，引擎已经替用户处理好消息等待，消息到达时会有OnMsg被触发

- bool Send(const unsigned char* pMsg, unsigned short uLength);
  发送数据
  返回值：
  当连接无效时，返回false

- void Close()
  关闭连接

- void Hold()当前版本(V1.20)中已经删除，被复制构造与析构取代，自动化管理引用计数与共享指针的释放
  保持对象，使用完，必须Free()
  用户在OnConnect OnMsg OnClose时候有机会得到该主机的指针
  退出以上方法后，指针就可能被引擎释放。

  但客户有可能需要保存该指针，在自己的业务使用，那么每次保存该指针，都必须Hold一次，让引擎知道有一个线程在访问该指针，每用完必须Free一次，相当于智能指针引用计数的概念

  ※使用案例
  案例1

```
OnConnect(NetHost *pClient)
{
    pClient->hold()
    pClient->send()
    pClient->free()
    接口说明说了，pClient在回调退出前，绝对安全，底层不会释放，hold free并不会引起错误，但不需要
}
```

  案例2

```
OnConnect(NetHost *pClient)
{
    连接建立，保存指针到map中，在业务处理中需要向该主机发送消息（转发、某玩家攻击该玩家等情况）时，使用
    pClient->hold()
    Lock map
    Map.insert(pClient->ID(),pClient);//这里pClient被保存了1次，hold了1次
    unlock map
}
OnCloseConnect (NetHost *pClient)
{
    连接断开，指针没有用了，从map删除，并free
    Lock map
```

pClient->free()
map.del(pClient->ID()) // pClient，从map删除， 1次hold了，free1次
unlock map
}
在以上前提下
错误方式1
OnMsg(NetHost *pClient)
{
攻击主机id 为1的玩家
Lock map
NetHost *otherHost = map.find(1)//这里第2次保存，没有hold， 还是1次
Unlock
使用otherHost，通知玩家受到攻击//如果这时玩家1正好退出游戏，并发生线程切换进入OnCloseConnect，指针被free
2种情况
1. 底层定时器检查指针状态的时间未到达，指针不会被释放，线程切换回OnMsg，发送通知，返回连接已断开，安全退出onMsg
2. 底层定时器正好到达检查时间点，发现指针的访问计数归0，将删除指针。otherHost变为野指针，线程切换回OnMsg，发送受到攻击通知时，访问野指针，系统崩溃
}
错误方式2
OnMsg(NetHost *pClient)
{
攻击主机id 为1的玩家
Lock map
NetHost *otherHost = map.find(1)//这里第2次保存，没有hold， 还是1次
Unlock
otherHost->hold()错误，情况同上，可能在hold时，已经是野指针
使用otherHost，通知玩家受到攻击
otherHost->free()

}

正确方式1
OnMsg(NetHost *pClient)
{
攻击主机id 为1的玩家
Lock map
NetHost *otherHost = map.find(1)//这里第2次保存，没有hold， 还是1次

使用otherHost，通知玩家受到攻击，即使线程切换到free线程，因为lock，free线程会被挂起，等待这里unlock，底层不会释放，安全

Unlock

退出OnMsg

}

正确方式2

OnMsg(NetHost *pClient)

{

攻击主机id 为1的玩家

Lock map

NetHost *otherHost = map.find(1)//这里第2次保存，没有hold， 还是1次

otherHost->hold()，同上在unlock之前，free线程不可能执行，底层不可能释放，

Unlock

使用otherHost，通知玩家受到攻击，已经hold2次，即使线程切换到free线程，访问计数还是大于1，底层不会释放，安全

otherHost->free()//使用完毕，释放访问，退出OnMsg

}

- void Free()当前版本(V1.20)中已经删除
  释放对象
  在完全不访问该指针后，Free()次数必须与Hold()相同，否则底层永远不释放该指针。

- bool IsServer()
  主机类型是一个服务
  NetServer调用Connect连接到一个服务器时，产生的NetHost是指向一个服务主机的，该方法返回true，否则返回false
- void InGroup(int groupID)
  放入某分组，同一个主机可多次调用该方法，放入多个分组
  与NetEngine::BroadcastMsg偶合
- void OutGroup(int groupID)
  从某分组删除
  与NetEngine::BroadcastMsg偶合

4 STNetServer

4.1 类说明

单线程版服务器基类
与NetServer用法相同

5 STNetHost

5.1 类说明

网络主机类
用NetHost类用法相同