狼人杀游戏类图框架

Last Update 2017/05/20 20:52

Update Log

将`ServerCenter`也视为`EventHandler`的子类，同时将`EventHandler`的机制做了部分修改。

ToDoList

1. Deadline 2017/05/31
2. 稍后画类图。
3. GUI的Qml/C++交叉编程样例。
4. 服务器、客户端网络接口。
5. 服务端简单调试工具。

背景知识

1. QT的SIGNAL-SLOT（信号槽）机制。

   详情可以参考这里。

2. 有关于类图。

   参考《设计模式可复用面向对象软件基础》Page249~251。

有关“变与不变”在当前设计模式下的若干问题

1.

服务器端

User

1. Description

- 表示一个用户。

2. Members
3. Methods

Spliter

1. Description

- 在一般的通信协议中，由于一次可能收到或者发出多条独立指令，添加分隔符将或以某种分割规

则将指令分隔开是必须的，这个Spliter的作用就是在某种通信规则下，将字符串划分为多条指令。

2. Members
3. Methods

- ```
  public virtual std::vector<QString> split(QString message) = 0;
  ```

ServerNetworkInterface

1. Description

- 服务端网络接口，服务器与客户端的所有通信操作均由这个来处理。
- ***以一个唯一的字符串Username来表示用户名而无需获知User类，达到了解耦的目的。***
- 由于使用了QT的Signal-Slots机制，事实上是异步运行的，需要特别注意。

2. Members

- ```
  private EventHandler* mainHandler;
  //一个指向服务器数据交换中心的指针。
  ```

- ```
  private std::map<QString, QTcpSocket*> userNameSocket;
  //一个用户名指向对应套接字的映射。
  ```

- ```
  private std::map<QTcpSocket*, ClientStatus*> socketClientStatus;
  //一个套接字指向对应客户端状态的映射。
  ```

- ```
  private Spliter* spliter;
  //基于当前通信协议的的字符串分割器。
  ```

3. Methods

- ```
  public void sendMessage(QString userName, QString message) {
      socketClientStatus[userNameSocket[userName]]->addSendString(message);
  }
  //上级要发指令啦。将指令存到对应的ClientStatus*中等待之后发送。
  ```

- ```
  private slots void startRead(QTcpSocket* tcpSocket) {
      //通过数据流处理，确认字符串s是已经读完整的字符串。
      std::vector<QString> afterSplit = spliter->split(s);
      QString userName = sockerClientStatus[tcpSocket]->userName;
      for (auto message : afterSplit)
          mainHandler->tryHandle(userName, message);
  }
  //接收到某个客户端发来的一串指令，移交给上级数据交换中心去分析。
  ```

ClientStatus

1. Description

- 对于与服务器连接的某一个客户端，保存与其通信必要的信息。

2. Members

- ```
  private QString userName;
  //显而易见就是用户名了。
  ```

- ```
  private QString nextSendString;
  //下一次要发给这个客户端的字符串，暂定使用JSON格式保存。
  ```

3. Methods

- ```
  public void addSendString(QString s) {
      nextSendString += s;
  }
  //将一条要发给客户端的指令保存下来接下来发送。
  ```

- ```
  public void clearString() {
      nextSendString = "";
  }
  //在保存的所有指令都已经开始发送之后，清空保存的指令。
  ```

- ```
  public QString getNextSendString() {
      return nextSendString;
  }
  //在保存的所有指令都已经开始发送之后，清空保存的指令。
  ```

EventHandler

1. Description

- 用来实际接受用户的指令，进行处理，并返回给此用户或者其他用户返回的信息。
- 使用责任链设计模式达成信息传递。

2. Members

- ```
  protected ServerNetworkInterface *networkInterface;
  //保存网络接口的指针用来交互
  ```

3. Methods

- ```
  private virtual bool canHandle(QString s) = 0;
  //返回这个handler能否处理用户的这条指令
  ```

- ```
  public void tryHandle(QString userName, QString message) {
      if (canHandle(message))
          handle(userName, message);
      else
          selectHandler(message)->tryHandle(userName, message);
  }
  ```

- ```
  private virtual void handle(QString userName, QString message) = 0;
  //进行实际处理
  ```

- ```
  private void sendMessage(QString userName, QString message) {
      networkInterface->sendMessage(userName, message);
  }
  //给客户端回信
  ```

- ```
  private virtual EventHandler* selectHandler(QString message) = 0;
  //根据信息情况返回自己内部的某个handler
  ```

ServerCenter [public EventHandler]

1. Description

- 服务器的数据交换中心。

2. Members

- ```
  EventHandler *roomManager;
  //管理现在在使用的那些房间。
  EventHandler *userDataBase;
  //管理用户数据
  //以上两个指针在初始化时或在第一次在selectHandler中用到时再使用new分配内存
  ```

3. Methods
  ```

- ```
  private bool canHandle(QString message) {}
  private EventHandler* selectHandler(QString s) {}
  private void handle(QString userName, QString message) {}
  //这三个函数按照EventHandler去实现
  ```

RoomManager [public EventHandler]

1. Description

- 用来管理房间。
- 事实上基本上一个`Room*[]`就行了...
- 通过`selectHandler`将锅甩给对应的`Room*`。

2. Members

-

3. Methods

-

Room [public EventHandler]

1. Description

- 将玩家组织在一起的基本单元。有玩家准备、房主开始游戏等功能。

2. Members

- ```
  private bool *ready;
  //玩家是否准备
  private Game *game;
  //游戏指针
  ```

3. Methods

- ```
  //在handle中调用，以下同
  private void userReady(QString userName) {
      ready[getUserPos(userName)] = true;
      //通过重载的sendMessage函数向所有玩家发送此玩家已准备的消息
      if (canStartGame()) {
          //向房主玩家发送“激活开始按钮”的消息放到ans中
      }
      return ans;
  }
  ```
  ```

- ```
  private void userUnReady(QString userName) {
      bool canStart = canStartGame();
      ready[getUserPos(userName)] = false;
      //向所有玩家发送此玩家取消准备的消息，通过重载的sendMessage函数
      if (canStart && !canStartGame()) {
          //向房主玩家发送"冻结开始按钮"的消息放到ans中
      }
  }
  ```

- ```
  private void enterRoom(QString userName) {
      //如果房间已经满了，向这个玩家返回房间已满的消息
      //否则向这个玩家返回进入房间的消息，位置分配为第一个未被占据的位置，并向其他玩家返回此玩家进入房间的消息
  }
  ```

- ```
  private void leaveRoom(QString, userName) {

  }
  ```

- ```
  private void startGame() {
      game = new Game();
      GameResult result = game->run();
      delete game;
  }
  //当接收到房主发来的“开始游戏“指令后在handle中调用此函数。
  ```

GameResult

1. Description

- 一个描述游戏结果的类。具体怎么写再说。

2. Members
3. Methods

Game [public EventHandler]

1. Despriction

- 服务器端游戏主逻辑。
- 主要通过`Round`的拆分来降低耦合度。
  - 在`Game`、`Round`中很可能还需要别的子类，请以可拓展性为最优先进行设计...

2. Members

- ```
  Round *round;
  //保存当前的Round的指针
  ```

3. Methods

- ```
  public result run() {
      round = new InitialRound();
      GameResult result;
      while (1) {
          result gameOver = round->run();
          if (gameOver != GameResult::gameRunning)
              return result;
          Round *temp = round->nextRound;
          delete round;
          round = temp;
          //防止内存泄漏
      }
  }
  //运行游戏。
  ```

Round [public EventHandler]

1. Description

- 使用之前谈论过的QT内置异步事件机制进行处理。
- 具体思想可以参加如下代码：

- ```
  class DayRound : public Round {
      Q_OBJECT //为了能够发送或接收信号必须写上
  public:
      DayRound() {}
      ~DayRound() {}

      //开始这个Round，返回游戏是否结束
      bool run() {

          //开始前的胜负判定

          nowStatus = 1;
          startAwaitSession(period1);
          //进行第一阶段的接包，等待至多period1毫秒
          /*
          这里进行第一阶段的处理
          调用server->sendMessage(username, message)给各客户端发消息
          */

          nowStatus = 2;
          startAwaitSession(period2);
  ```
  ```

```
        //进行第二阶段接包，等待之多period2毫秒
        /*
        这里进行第二阶段处理
        调用server->sendMessage(username, message)给各客户端发消息
        */

        //结束后的胜负判定

    }
    void handle(QStrin userName, QString message) {
        //按照nowStatus以及传来的信息接收包。
        if (allReceived) {//进行判断，如果所有需要的包都已经接受到了：
            emit receiveOK(); //自己发出receiveOK()的信号
        }
    }
signal:
    void receiveOk();
private:
    int nowStatus;
    //表示现在进入这个Round的什么阶段了
    GameServer *server;
    Round *nextRound;
    //为了能够传递信息必须要有这个指针了...

    //开始一个至多长达msec毫秒的异步等待
    void startAwaitSession(int msec) {
        QTimer *timer = new QTimer(this);
        //QT中的计时器
        timer.setInterval(msec);
        //将计时器的周期设为msec毫秒
        QEventloop *loop = new QEventloop(this);
        //QT内置的异步事件循环
        timer->start();
        //计时器开始计时
        connect(timer, SIGNAL(timeout()), loop, SLOT(quit()));
        //进行信号槽连接：当timer到时就使得事件循环退出
        connect(this, SIGNAL(receiveOK()), loop, SLOT(quit()));
        //进行信号槽连接：当类本身发出receiveOK()的信号时，事件循环退出
        loop.exec();
        //事件循环开始运行。这行代码是异步的，不会占用cpu，但是如果这行代码没有运行完毕，下面的代码无法开始运行。
    }
};
```

-

2. Members

-

3. Methods

- ```
  private Round* nextRound() {}
  //给接下来的一个Round分配内存并返回
  ```
-

客户端

ClientNetworkInterface

1. Description

- 客户端用来与服务器通信的类。

GUI

使用Qt Quick技术进行编程。

以上两者使用QmlI\C++交叉编程，具体实现见《Qt Quick 核心编程》Chapter11。
  ```