

“生气斗地主”设计文档

© 聂鼎宜 (Delphynium Nie), no.2019011346

//这是基于 Qt 编写的网络游戏工程“生气斗地主”的设计说明文档。

//经测试，游戏在 Qt 5.15.0，Clang 11.0，macOS 10.15.16 环境下能够正常编译，图形界面及各项功能运作良好。

//由于整个工程代码量已经超过千行，出于成体系的考量，本文将在介绍完核心设计理念后，分文件地介绍各项功能的实现。

1 核心设计理念

网络游戏的核心是通信。由于甲方要求中明文禁止了中心化的通信结构或另起进程，故无法各自编写服务器和客户端。

因此，该游戏参考了传统网络游戏中的“房主”概念，将服务器以**另起线程**的模式整合在客户端内。只有开设房间的玩家的客户端才会实例化这一线程，他也就成为游戏的“房主”，允许其他玩家加入自己的房间。

房主与游戏房间是绑定的，房主退出游戏或断开连接将导致房间的解散；但房间内的其他玩家可以动态的，即允许玩家的退出和新玩家的加入。

斗地主作为一款棋牌类游戏，在抽象层面上，作为服务器的线程自然地设计为具有 "Dealer" 的一切职责和功能，负责监督游戏进度，并根据游戏规则分发、接收、处理事件（比如发牌、判定出牌合法性）。而客户端内除服务器线程外的其他组件均只负责图形界面的渲染与玩家输入信号的监听、收集、发送。

注意，传统的网络游戏客户端的职责并不应该比服务器少，比如 3D 网络游戏的客户端应整合了物理引擎等组件来限制玩家的行为；但在“生气斗地主”中，客户端并不负责在交送数据到服务器前判定玩家操作的合法性（最典型的，出牌的合法性），而仅仅诚实地发送数据，将一切判定交给服务器来做。从抽象意义上讲，也就是默认玩家是不具备判断力的“婴儿”，而服务器是其“监护人”，全权监控其行为。“婴儿”无论做了什么都要向“监护人”汇报，“监护人”命令“婴儿”做什么“婴儿”就要做什么。

这就是“生气斗地主”的核心设计理念。





2 CardLabel 类及其子类 HandLabel

2.1 CardLabel 类

CardLabel 类的职责是抽象“扑克牌”这一概念。在 "cardlabel.h" 中，CardLabel 类的定义如下：

```
1 class CardLabel: public QLabel{
2
3     Q_OBJECT
4
5 public:
6     CardLabel(int id, QWidget* parent);
7     int getCardId();
8     static std::pair<int, int> id2Info(int id);
9     static QPixmap getPixmap(int id);
10
11 private:
12     int key;
13     int suit;
14     int id;
15     QPixmap image;
16 };
```

现实中，扑克牌是成套出现的，并且每张牌有三个要素：花色、点数、图案。鉴于其与 "label" 概念的相似性，CardLabel 将继承自 QLabel 类。

CardLabel 类定义了一个整型 suit 来描述牌的花色。取值范围为 [0, 4)，依次对应 、、、。

虽然点数为 A~K 的扑克牌内蕴了一个整型 (1~13)，但斗地主的规则中，A 和 2 是**压倒** 3~K 的。因此，CardLabel 中的 key 是**基于这一压倒规则的点数**，取值范围为 [0, 15)，依次对应 3~K、A~2、*JOKER*、*JOKER*。

为了描述“一副扑克牌”这一 54 元集合，CardLabel 类定义了一个整型 id 来区分不同的元素（牌）。它的取值范围是 [0, 54)。为了便于先点数、后花色的排序， 3~ 2 依次对应 0~51，*JOKER* 对应 52，*JOKER* 对应 53。

id2Info 方法的作用是接收一个上述 id，返回该 id 对应的 key 和 suit。CardLabel 的构造函数就是仅接收卡牌的 id，再调用该方法完成构造的。

QPixmap 型 image 在构造时就已从资源库里 load 了牌面的 .jpg 图像，在对象有必要在图形界面显示出来时，image 将会通过 setPixmap 方法被 CardLabel 对象加载。

getPixmap 方法的作用就是接收一个 id，自动计算该 id 对应卡面在资源库中的路径并生成对应卡面的 QPixmap。当传入 id 为 -1 时，返回的 QPixmap 是扑克牌的卡背。

至此，概念意义上的扑克牌已经完成抽象... 但还缺点什么。

2.2 HandLabel 类

为了进一步抽象现实中“手牌”的概念，定义继承自 CardLabel 类的 HandLabel 类：

```
1 class HandLabel: public CardLabel{
2
3     Q_OBJECT
4
5 public:
6     HandLabel(int id, QWidget* parent);
7     void setUp();
8     void setDown();
9     void setUncovered();
10    bool isUp();
11    static void isMyTurn(bool);
12
13 protected:
14     void mousePressEvent(QMouseEvent *);
15
16 private:
17     static bool my_turn;
18     bool covered;
19     bool is_up;
20 };
```

在 CardLabel 的基础上，HandLabel 新增的核心功能是**监听玩家的选中与取消选中操作**。为此，HandLabel 中的 mousePressEvent 接口被重写，使其一旦接收到**有效的鼠标点击事件**时能够作出视觉响应。“有效”体现在两个方面：

- 是否是本人回合。只有当私人 bool 型成员 my_turn 为 true 时，mousePressEvent 才会进行监听，否则直接返回。静态 isMyTurn 方法是 my_turn 的接口，当游戏来到本人回合时，isMyTurn 被调用，my_turn 被设为 true；离开本人回合时反之。
- 是否点击在卡牌的**可视区域**。由于手牌存在相互重叠，故这一判定是必要的。mousePressEvent 在重写中实现了 event 的 pos 是否在有效区域的判定，这个判定的规则取决于私有 bool 型成员 is_up 和私有 bool 型成员 covered：当 is_up 为 true 时（说明卡牌被选中而且显示位置上移），卡牌的上部区域被纳入有效区域；当 covered 为 true 时（说明这张牌不在顶层图层），只有卡牌的左部区域有效，否则卡牌的整个卡面都有效。

setUp 和 setDown 方法是 is_up 的接口，用来管理 is_up 状态。当 mousePressEvent 接到有效的点击事件后，根据当前的 is_up 状态来调用相反的方法。

isUp 方法提供了访问 is_up 的外部接口。

3 MainWindow类和客户端工作流程

3.1 MainWindow 类

MainWindow 类在 "mainwindow.h" 中的定义如下:

```
1 class MainWindow : public QMainWindow{
2
3     Q_OBJECT
4
5 public:
6     explicit MainWindow(QWidget *parent = nullptr);
7     ~MainWindow();
8
9 private:
10     Ui::MainWindow *ui;
11
12     MyDialog *dialog;
13     ServerThread *server;
14     QTcpSocket *server_socket;
15
16     QPushButton *start_button;
17     QPushButton *deal_button;
18     QPushButton *seat_button;
19     QPushButton *pass_button;
20     QPushButton *play_button;
21     QPushButton *jiaodizhu_button;
22     QPushButton *bujiao_button;
23     QLabel *played_illegally_label;
24     QLabel *player_image[3];
25     QLabel *player_seated_label[3];
26     QLabel *player_jiaodizhu_label[3];
27     QLabel *player_pass_label[3];
28     QLabel *player_hands_number_icon[3];
29     QLabel *player_hands_number_label[3];
30     int player_hands_number[3];
31     QLabel *player_yaobuqi_label[3];
32     QLabel *player_identity_label[3];
33     QPixmap nongmin_image;
34     QPixmap dizhu_image;
35
36     QVector<HandLabel *> player_hands_labels;
37     QVector<int> player_hands_array;
38     CardLabel * bottom_labels[3];
39     QVector<CardLabel *> player_played_labels[3];
40     QVector<int> player_played_array[3];
41
42     bool is_host;
```

```

43     int player_number;
44
45     void initScene();
46     void clearScene();
47     void updateHandsLabels();
48     void updatePlayedLabels(int);
49     int getPosition(int);
50
51     void sendToServer(QString);
52
53 protected slots:
54     void showDialog();
55     void createServer(QString);
56     void joinServer(QString, QString);
57     void readFromServer();//client reads
58     void readError(QAbstractSocket::SocketError);
59     void seatButtonClicked();
60     void dealButtonClicked();
61     void jiaodizhuButtonClicked();
62     void bujiaoButtonClicked();
63     void playButtonClicked();
64     void passButtonClicked();
65 };

```

MainWindow 类封装内容的主体是游戏主窗口的控件和图形部分，包括所有的按钮（及其 clicked 信号连接到的槽函数）、标签和图形素材，能够正确地显示左、右、底部三个玩家的各种状态。**注意，虽然代码中没有显式指明，实际上三个位置对应着三个位置标号 i：底部为 0 号位（即玩家自己），右部为 1 号位，左部为 2 号位。所有长度为 3 的图形部件的数组都满足这样的对应关系。**

MyDialog 类的指针 dialog 指向的对象是向玩家提供“创建房间”和“加入房间”选择的对话框，其内容将在第 5 节进行说明。

ServerThread 类的指针 server 所指的对象就是被封装在线程内的游戏服务器。服务器与客户端之间存在一套基于 socket 的通信机制，将在第 4 节进行详细说明。

QTcpSocket 类的指针 server_socket 指向与 server 进行通信的 socket。

整型 QVector 容器 player_hands_array 存储着玩家当前拥有的手牌的 id。玩家得到初始牌、得到底牌和打出手牌的底层变化都是这个向量的变化。

HandLabel * 型的 QVector 容器 player_hands_labels 存储着指向玩家当前所有的手牌的指针。这些手牌将被显示在窗口底部。

updateHandsLabels 方法的作用是提供从 player_hands_array 到 player_hands_labels 的一个映射，从抽象意义上讲，它的作用是将数据层面的手牌**渲染**为图形层面手牌以便与玩家进行交互。每次 player_hands_array 发生任何变化时，updateHandsLabels 都应该及时被调用。

CardLabel * 数组 bottom_labels 指向游戏开始后始终显示在窗口上方的三张底牌。

整型 QVector 容器数组 player_played_array 的第 i 位上存储着 i 号位玩家**这回合打出的牌的 id**。当他打出手牌时，向量更新；当他的下一回合开始时，向量将被清空。

CardLabel * 型的 QVector 容器数组 player_played_labels 的第 i 位存储着指向 i 号位玩家这回合打出的所有牌。这些牌将被显示在窗口内各玩家对应的位置上。

updatePlayedLabels 方法提供了从 player_played_array 到 player_hands_labels 的映射，其抽象意义的作用是**渲染**玩家打出的手牌。每当 i 号位玩家打出手牌或进入他的回合，updatePlayedLabels 将被调用，并传入 i 作为参数，刷新 i 号位的“桌面”的显示。

bool 型 is_host 成员指示该玩家是否是房主，即该客户端是否实例化了 ServerThread 对象。它将决定玩家是否有发送发牌请求（即开始游戏的信号）的权利。

int 型 player_number 存储**玩家在服务器上的编号**。注意，虽然这个编号的取值也是 [0,3)，但它与位置标号并不相等：**在玩家的游戏窗口内，他自己的位置永远是 0 号位，但在服务器上他不一定是 0 号玩家**。当玩家连接上服务器时，由服务器来分发这个编号。服务器对玩家的编号满足：房主是 0 号玩家，0 号玩家是 1 号玩家的上家，1 号玩家是 2 号玩家的上家。服务器的所有广播内容中，涉及到特定玩家发生的事件，引用的都是这位玩家在服务器上的编号。客户端接到信号后，需要使用 getPosition 方法推算出这个编号指的是本地的几号位的玩家（由于玩家的上下家关系是确定的，且客户端知道自己的编号，所以推算可以在本地完成）再进一步处理这个事件。

整型数组 player_hands_number 维护着三位玩家的手牌数量信息。每次该数组发生变化时，更新对应玩家的手牌数量显示。

sendToServer 方法可以便利地将一个 QString 字符串发送到服务器，被广泛使用在与服务器的交互。

initScene 方法的作用是初始化游戏界面，即还没有玩家加入房间的状态。

3.2 客户端工作流程

客户端的工作流程可以概括如下：

1. 客户端启动时，初始化为开始界面，提供“一键破产”按钮 (start_button)，连接到槽函数 showDialog。玩家点击后，弹出 dialog 引导玩家从“创建房间”或“加入房间”中选择其一并键入 ip 与端口。
2. 如果玩家选择创建房间，则构造 ServerThread 对象 server，server 内维护了一个 QTcpServer，此时开始监听。同时，玩家的客户端将通过 QTcpSocket 向 server 建立网络连接。否则，玩家选择加入房间，如果房间有效，客户端也将通过 QTcpSocket 与 server 连接。
3. 当 server 接受到新的连接请求时，为该连接分配一个 [0, 3) 的编号并发送给客户端，并提供已经加入房间的其他玩家的信息，指挥目标进行游戏界面的初始化。同时，server 向其它玩家播报新玩家的进入。客户端接到信息，将在对应位置初始化玩家的状态。
4. 准备阶段，房主的界面上“发牌”按钮处于可见状态，而其他玩家的“坐下”按钮 (seat_button) 处于可用状态。当玩家坐下，将向服务器发送信息，服务器接到信息再向所有人广播。当三人都完成“坐下”，服务器向房主发送信息，允许发牌。此时“发牌按钮”将可

用。

5. 房主点击“发牌”后，向服务器发送一个信息。服务器接到信息，便运行随机数算法进行初始手牌的分配，再将每位玩家的手牌信息（以 `id` 形式存储）封包、分别发送给每位玩家。客户端接到包，用包中的信息初始化自己的手牌。此时服务器进一步通过随机数决定叫地主阶段的起点，并向该玩家发送信号。
6. 最先叫地主的玩家客户端收到信号，“叫地主”按钮 (`jiaodizhu_button`) 和“不叫”按钮 (`bujiao_button`) 处于可用状态。玩家决定后，向服务器发送对应信息，服务器处理后接着向下一位玩家发送信号，直到完成叫地主阶段。
7. 服务器决定地主后，向地主玩家发送信号并将底牌信息封包公布给所有玩家。玩家客户端接到包后，用它来设定窗口上方的底牌。如果同时接到了被决定为地主的信号，底牌将同时加入手牌。此时更新玩家手牌数信息。游戏正式开始。所有玩家的“出牌”按钮 (`play_button`) 和“不出” (`pass_button`) 被设为可见。服务器的设计考虑到了“无法不出牌”的情况，这包括连续两位玩家不出后的回合和地主玩家的第一个回合，因此此时地主玩家的“不出”按钮处于不可用状态。
8. 玩家若选中了数张牌设为 `is_up` 状态，然后点击“出牌”按钮，客户端将会遍历 `player_hands_labels` 数组，获取所有 `is_up` 的手牌并将它们的 `id` 封包、发送至服务器。服务器接到包后判断出牌的合法性。若合法，向所有玩家广播该玩家的出牌信息并更新相应容器，然后向下一玩家发送回合到来的信息；若不合法，向该玩家反馈信息。玩家接到信息后，更新相应的容器与可视部件的状态。若玩家选择“不出”，服务器在接到信息后向所有玩家广播，然后进行下一回合。
9. 直到某次出牌合法后，服务器判定出牌玩家手牌数归零，游戏进入终止阶段。服务器在判定胜负后向所有玩家发送其胜/负信息，客户端接到信息后弹出相应的窗口，并指引玩家选择退出房间/再来一局。
10. 若玩家选择再来一局，客户端会向服务器发送一个信号，服务器将指引客户端进行准备界面的初始化。若玩家选择退出，则会直接断开与 `server` 的连接并退回开始界面。如果是房主退出，`server` 将析构，其他客户端将收到 `SocketError`，弹出提示信息后退回开始界面，房间就此解散。

如此，客户端完成了一局游戏循环。

4 ServerThread 类

4.1 ServerThread 类

`ServerThread` 类在 "`serverthread.h`" 中的定义如下：

```
1 class ServerThread: public QThread{
2
3     Q_OBJECT
4
5 public:
6     ServerThread(QString port ,QObject* parent);
7
8 protected:
9     void run();
10
11 private:
12     QTcpServer *listen_socket;
```

```
13     QTcpSocket *client_socket[3];
14
15     QVector<int> player_hands[3];
16     QVector<int> card_pile;
17
18     enum Pattern {Anything,
19                   DanZhang,
20                   YiDui,
21                   SanDai,
22                   ShunZi,
23                   LianDui,
24                   SiDaiEr,
25                   FeiJi,
26                   ZhaDan,
27                   WangZha} current_playing_pattern;
28
29     int current_playing_key;
30     int current_playing_scale;
31     int current_playing_attach_scale;
32     int current_dizhu;
33     bool player_is_connected[3];
34     bool player_is_seated[3];
35     int pass_count;
36     int decide_count;
37
38     void initGame();
39     void startDeal();
40     void decideDizhu(int);
41     void nextTurn(int, bool);
42     void gameOver(int);
43
44     void broadcastAll(QString);
45     void broadcast(int, QString);
46     void broadcastReadyInfo(int);
47
48     bool judgeLegality(QVector<int> &);
49     bool judgeWangZha(int *);
50     int judgeZhaDan(int *);
51     int judgeDanZhang(int *);
52     int judgeYiDui(int *);
53     std::pair<int, int> judgeSanDai(int *);
54     std::pair<int, int> judgeShunZi(int *);
55     std::pair<int, int> judgeLianDui(int *);
56     std::pair<int, int> judgeSiDaiEr(int *);
57     std::tuple<int, int, int> judgeFeiJi(int *);
58
59 protected slots:
60     void acceptConnection();
61     void handleDisconnected();
```



```

62     void handleRead();
63 };
64

```

`listen_socket` 和 `client_socket` 分别是用来监听的 `QTcpServer` 和与三名玩家的客户端通信的 `QTcpSocket`。

`player_hands` 数组以 `id` 的形式存储三名玩家的手牌信息。`card_pile` 则被初始化为 `[0, 54)` 的数组，发牌的算法大致是每次从中随机抽出一张牌，轮流发给每位玩家，直到 `card_pile` 余下 3 个元素（底牌）。

`enum Pattern` 是对牌型的定义，成员变量 `current_playing_pattern` 记录了游戏当前正在出的牌型。若该回合为不可 `pass` 的，`current_playing_pattern` 将被设为 `Anything`。

`current_playing_key`、`current_playing_scale`、`current_playing_attach_scale` 三个整型成员共同描述当前桌面最顶层的牌（组）即最近一次玩家出的牌的特征。它们的规定如下：

- 单张：`key` = 这张牌的 `key`；另两者无意义。
- 一对：`key` = 这对牌的 `key`；另两者无意义。
- 三带：`key` = 成三张的牌的 `key`；`scale` 无意义；`attach scale` = 0（三不带）/ 1（三带一）/ 2（三带一对）。
- 顺子：`key` = 最小的那张牌的 `key`；`scale` = 顺子的长度（张数）；`attach scale` 无意义。
- 连对：`key` = 最小的那对牌的 `key`；`scale` = 连对的长度（对数）；`attach scale` 无意义。
- 四带二：`key` = 成四张的牌的 `key`；`scale` 无意义；`attach scale` = 1（四带两张）/ 2（四带两对）。
- 飞机：`key` = 飞机的主体中最小的成三张的牌的 `key`；`scale` = 飞机主体的长度（三张一组的组数）；`attach scale` = 0（不带）/ 1（每三张带一张）/ 2（每三张带一对）。
- 炸弹：`key` = 这个炸弹的 `key`；另两者无意义。
- 王炸：三者均无意义（王炸唯一）。

`current_dizhu` 用来在决定地主阶段暂存当前地主的编号。当游戏已经开始，`current_dizhu` 恒为本局游戏的地主。

`player_is_connected` 和 `player_is_seated` 用来记录（准备阶段）玩家的连接情况和准备情况。同时，`player_is_connected` 等效标记了哪些 `client_socket` 是可用的。

`pass_count` 和 `decide_count` 分别记录玩家连续“不要”的次数和已经“叫地主”/“不叫”的次数。当 `pass_count` 来到 2 时，归零并设下一回合为不可跳过的。当 `decide_count` 来到 3，归零并结束叫地主阶段，决定地主。

`initGame` 方法可以初始化对局。`startDeal` 是发牌函数。`decideDizhu` 接收一个整型参数 `player`，向 `player` 号玩家发送竞选地主的信息。

`nextTurn` 方法用来推进回合。其中，整型参数是下一回合出牌的玩家编号，`bool` 型参数设置该回合能否跳过（“不出”）。

`gameOver` 方法传入手牌归零的玩家编号，作用是决定胜者、发送相应信息。

broadcast、broadcastAll 是两个广播函数，分别能做到向特定玩家播报信息、向全体玩家播报信息。broadcastReadyInfo 仅在新玩家加入时调用，向新玩家广播目前房间内其他玩家的状态。

judgeLegality 方法传入一个牌的 id 组成的 QVector，用来判断出牌的合法性，返回一个表示合法/不合法的 bool 值。所有以 judge 开头的方法都是判定合法性功能的一部分，将在 4.3 详细解释。

槽函数分别处理新连接信号、断开连接信号和 readyRead 信号。其中 handleRead 方法的设计是游戏通信的核心，将在下一节详细解释。

4.2 通信协议

考虑到斗地主游戏的通信中，包的体量很小，收发也不算频繁，故通信协议的设计采用简单的**关键词 + 分隔符**模式。

在服务器、客户端向对方发送信息时，信息的模式统一为“keyword + 参数 + \n”。服务器和客户端在处理 readyRead 信号时，均调用 QTcpSocket 的 readLine 方法逐行读取，直到 canReadLine 方法返回 false。将读取到的流转入一个 QTextStream 对象内，便可以用流运算符方便地提取关键词。对关键词进行识别，并据此决定参数的读法以及响应的行为，这就是客户端与服务器间的通信模式。

注意，虽然 QThread 类在常规使用中经常通过信号-槽与主窗口交互，但这显然不符合中心化通信的模式。在实际运作中，无论玩家是否是房主，客户端（即 MainWindow 相关的代码）均是通过网络通信与服务器（即 ServerThread 相关的代码）进行交互的。

4.3 规则判定流程

在服务器接到玩家出牌信息的包时，先将玩家出的牌的 id 整合在一个 int 型的 QVector 里，然后将其传入 judgeLegality 方法（这一步发生在 handleRead 方法中）。

judgeLegality 接到数据后，首先做的事是判断 current_playing_pattern 是否为 WangZha。如果是，一切都是徒劳，可以直接返回 false。

否则，judgeLegality 首先将 QVector 中的内容“倾倒”进一个 int 型 count 数组里——这个数组长度为 15，count[0]~count[14] 依次记录 3~2、*JOKER*、*JOKER* 的张数。各 judge 函数接收的参数就是指向这个数组的指针。

先解释一下各 judge 函数的运作流程，再接着解释 judgeLegality 的逻辑：

- judgeDanZhang，如果判断为单张，返回其 key；如果判断为不是单张，返回 -1。接收到 count 数组后，首先从大到小遍历，寻找 count[i] > 0 的牌 i。如果 count[i] > 1，返回 -1，否则，继续遍历，确保没有其他牌，然后返回 i。
- judgeYiDui，如果判断为一对，返回其 key；如果判断为不是一对，返回 -1。接收到 count 数组后，首先从大到小遍历，寻找 count[i] > 0 的牌 i。如果 count[i] != 2，返回 -1，否则，继续遍历，确保没有其他牌，然后返回 i。
- judgeSanDai，如果判断为三带（三不带/三带一/三带一对），返回 (key, attach scale)，否则返回 (-1, 0)。接收到 count 数组后，先遍历寻找成三张的牌 i（若没找到返回 (-1, 0)），然后从小到大遍历，找不是 i 的第一张牌 j（如果没找到，返回 (i, 0)），判断

count[j]，分 1、2、>2 三种情况，前两者在确保没有其他牌后分别返回 (i, 1)、(i, 2)，否则返回 (-1, 0)。

- judgeShunZi，如果判断为顺子返回 (key, scale)，否则返回 (-1, 0)。接收到 count 数组后，从小到大遍历直到遇到第一张的牌 i，首先 count[i] == 1，并且从 i 开始至少有连续五张单张，并且后续不再有其他牌，方可保证为顺子。
- judgeLianDui，规则与 judgeShunZi 基本类似，不过寻找的是成对的牌。不再赘述。
- judgeSiDaiEr，规则与 judgeSanDai 基本类似，不过寻找的是成四张的牌，不再赘述。
- judgeFeiJi，如果判断为飞机，返回 (key, scale, attach scale)，否则返回 (-1, 0, 0)。接收到 count 数组后，首先从小到大寻找第一组成三张的牌 i，然后继续搜索找出飞机主体的长度，然后先判断剩下的牌数是否被飞机长度整除，如果是，再判断牌型是否满足成单或成对，最后返回相应值。
- judgeZhaDan，规则与 judgeYiDui 基本类似，不过寻找的是成四张的牌，不再赘述。
- judgeWangZha，如果判断为王炸，返回 true，否则返回 false。

judgeLegality 在工作中，会优先调用 judgeWangZha，因为王炸随时都可以出。

其后，judgeLegality 先判断 current_playing_pattern 是否为炸弹，若否，则进一步调用 judgeZhaDan，因为在该情况下炸弹一定可以出。

如果不是炸弹，则依次讨论 current_playing_pattern 的值。

- 若为 Anything，依次调用所有的 judge 函数，直到找到返回值（的首位）>= 0 的牌型，然后将 current_playing_pattern、current_playing_key、current_playing_scale、current_playing_attach_scale 置为相应值，最后返回 true。若没找到，返回 false。
- 若为 DanZhang，调用 judgeDanZhang，判断返回值是否 >= 0，如果是，则更新 current_playing_key，最后返回 true。否则返回 false。
- 其它情况原理类似，都是判断返回值首位是否 >= 0，再维护四个成员变量，最后返回 true/false。

这就是 judgeLegality 即规则判定的流程。

5 MyDialog 类

"mydialog.h"中，MyDialog 类定义如下：

```
1 class MyDialog: public QDialog{
2
3     Q_OBJECT
4
5 public:
6     MyDialog(QWidget *);
7
8 private:
9     QLineEdit *ip_input;
10    QLineEdit *port_input;
11    QRadioButton *create_a_room;
12    QRadioButton *join_a_room;
13
14 protected slots:
```

```
15     void setDone();
16 signals:
17     void connectInfo(QString ,QString);
18     void hostInfo(QString);
19 };
```

一言以蔽之，`connectInfo` 和 `hostInfo` 是两个人为定义的信号，在 `MainWindow` 中他们分别与 `joinServer` 和 `createServer` 槽连接，并将 `QLineEdit` 中的 `ip` 和端口信息传给它们，使后续连接的建立成为可能。

以上就是“生气斗地主”设计说明的全部内容。