

# 简易的国际象棋双人对战程序设计文档

---

## 程序介绍

### 现实背景

国际象棋是世界上一个古老的棋种。据现有史料记载，国际象棋的发展历史已将近2000年。关于它的起源，有多种不同的说法，诸如起源于古印度、中国、阿拉伯国家等。

国际象棋分为黑白两方共32枚，每方各16枚；棋盘为正方形，由64个黑白（深色与浅色）相间的格子组成。每方有王、后、象、车、马、兵六种棋子，不同棋子走子和吃子的方法不同，不再赘述。

### 设计目的

使用Qt自带的Socket编程，实现客户端和服务端一对一的网络对战，正确复现国际象棋的规则。在基本的走子和吃子回合制下，要求有：

对局中，出现下列情况之一，本方算输，对方赢：

- 1) 己方王被将死（不允许送吃）； 2) 己方发出认输请求； 3) 己方走棋超出步时限制；

实现兵生变，用户可选择生变为的棋子。

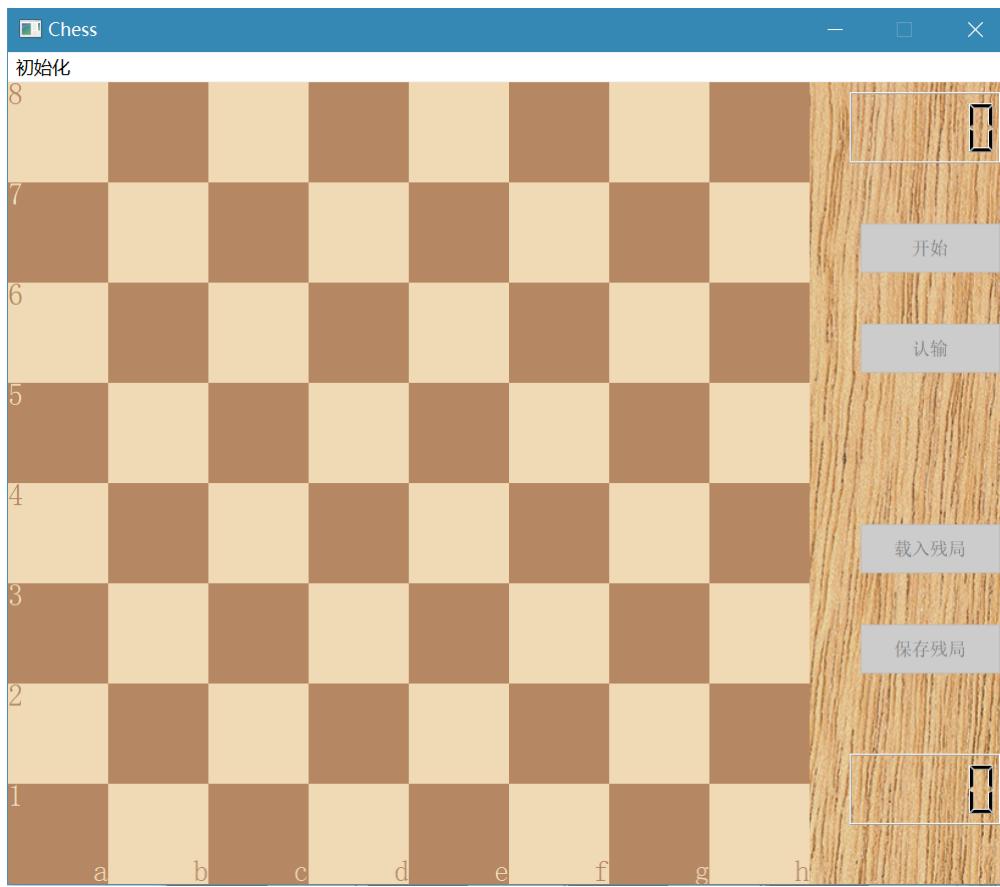
加入逼和和车马易位功能。车马易位不考虑车和马是否移动过，只要在固定位置，满足车马易位的其他条件即可进行易位。

一局结束时（胜负，逼和）在两端弹出对话框告知结果。

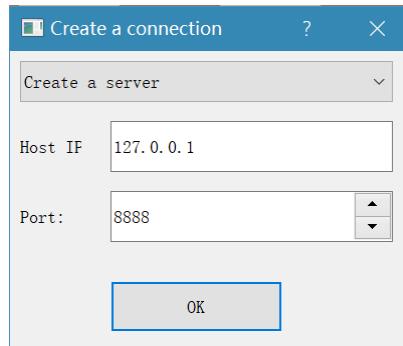
加入残局载入和保存功能。

### 程序使用方法

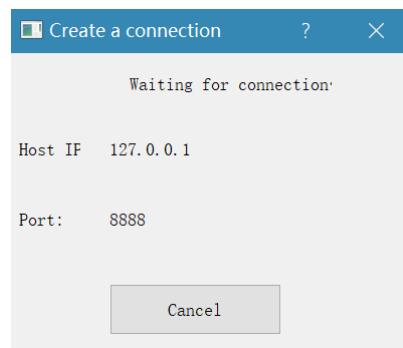
程序初始界面如下



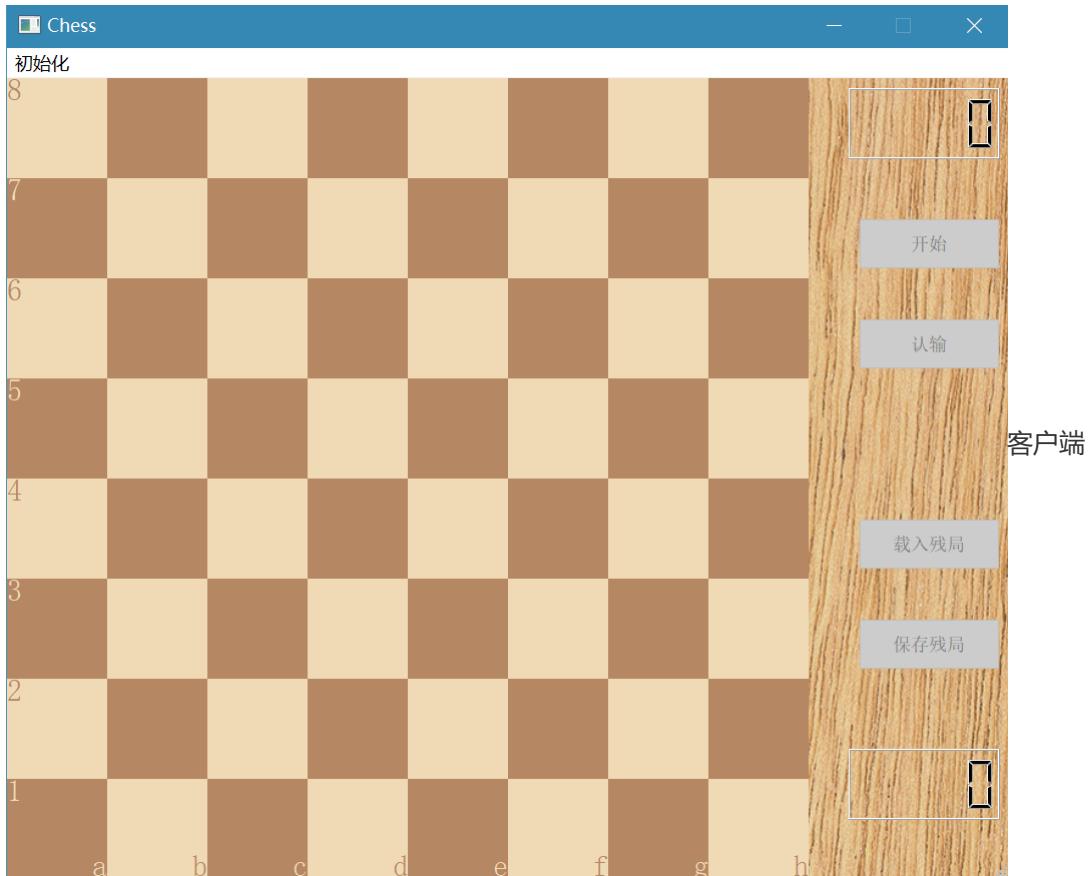
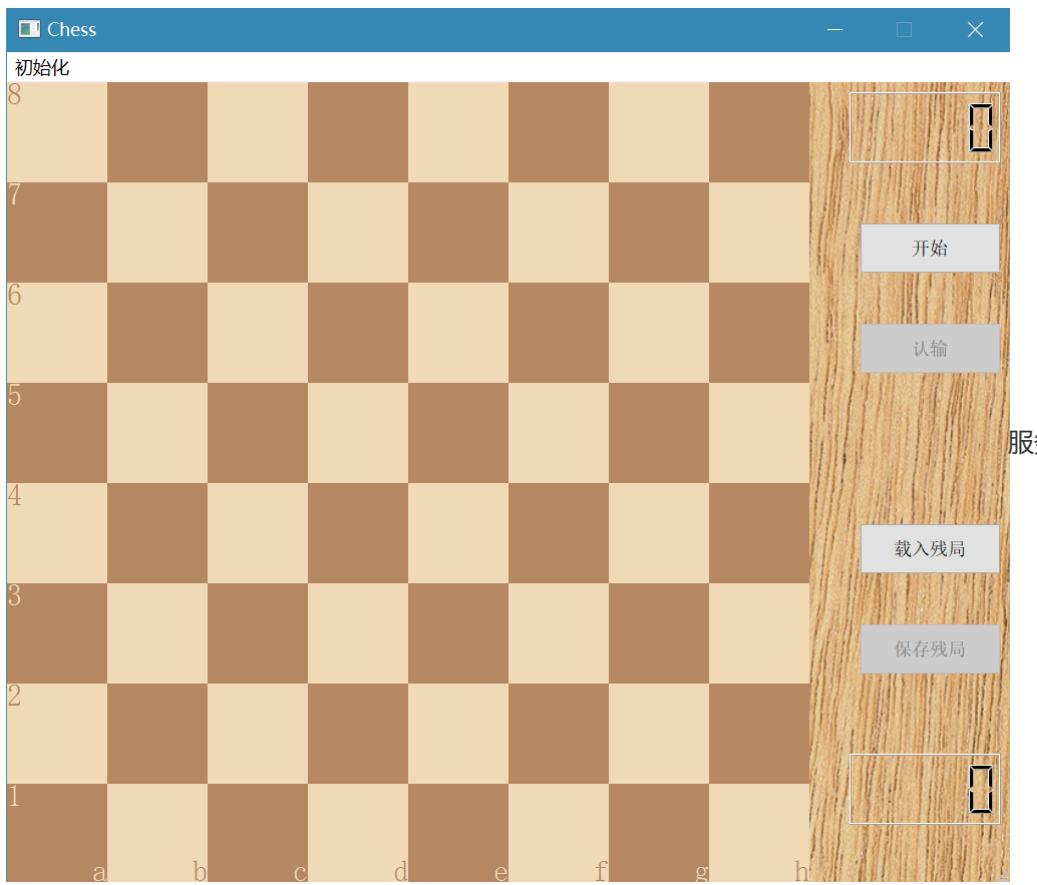
本程序采用客户端和服务端集成的设计，点击初始化—连接，用户自主选择创建客户端抑或服务端，以及ip地址和端口号。



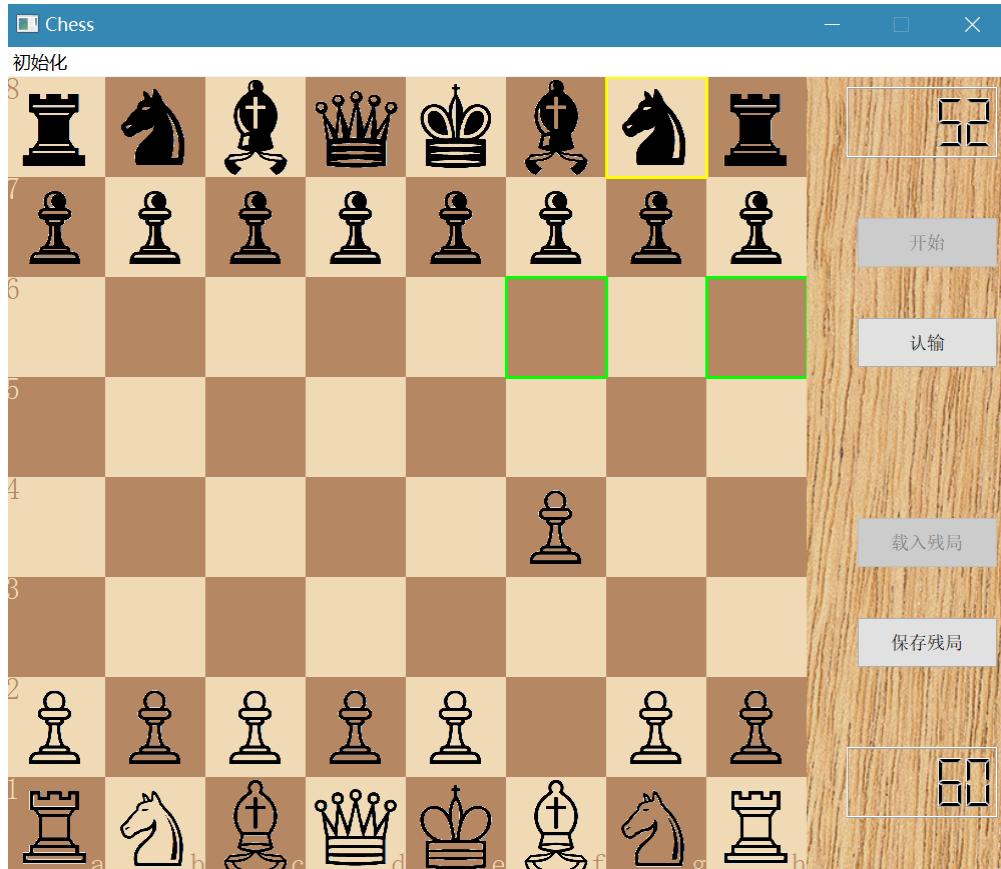
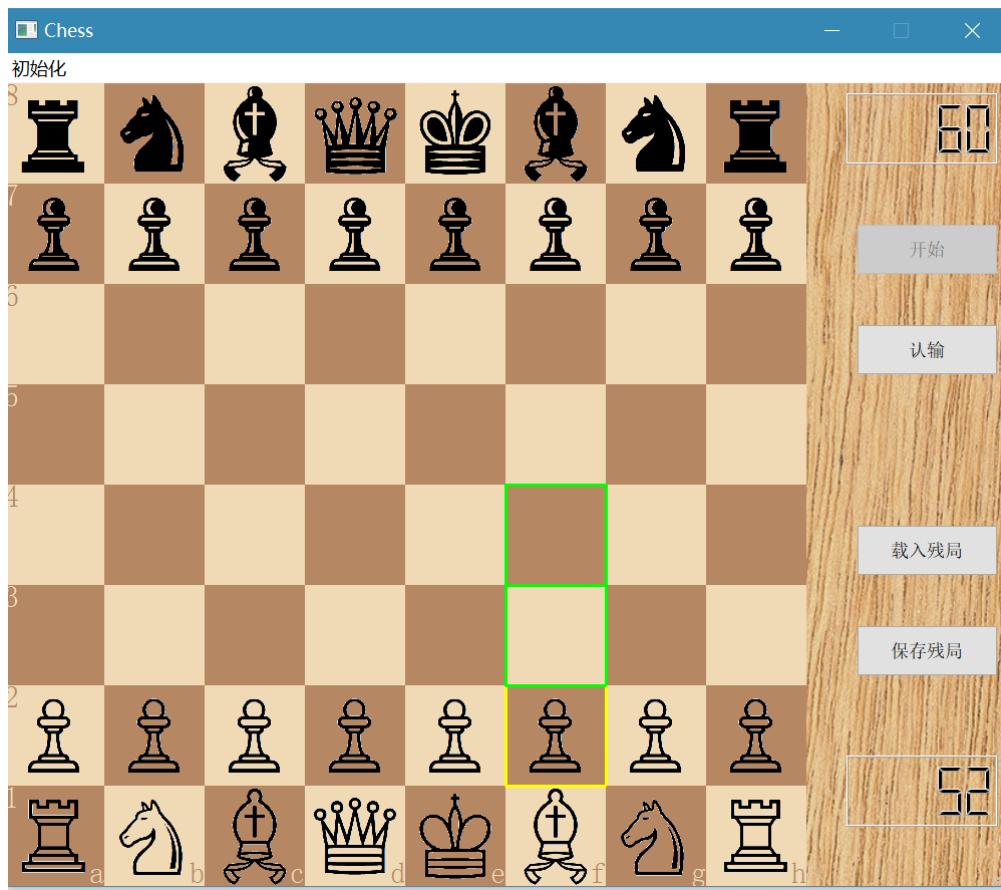
选择创建服务端，点击OK，处于等待连接状态；可随时取消等待。



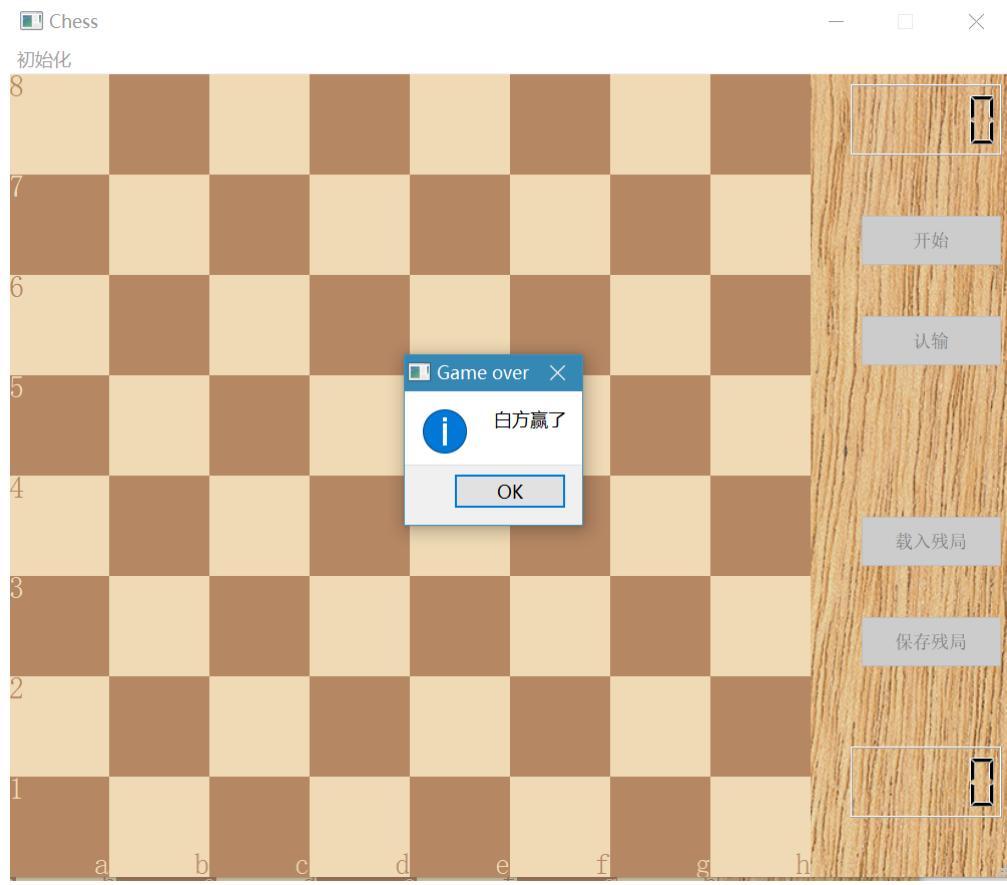
再开一个程序，选择创建客户端，设置对应端口号，点击OK连接服务端。这时客户端和服务端的创建连接窗口销毁，代表二者成功建立连接。这时两端的界面分别变为



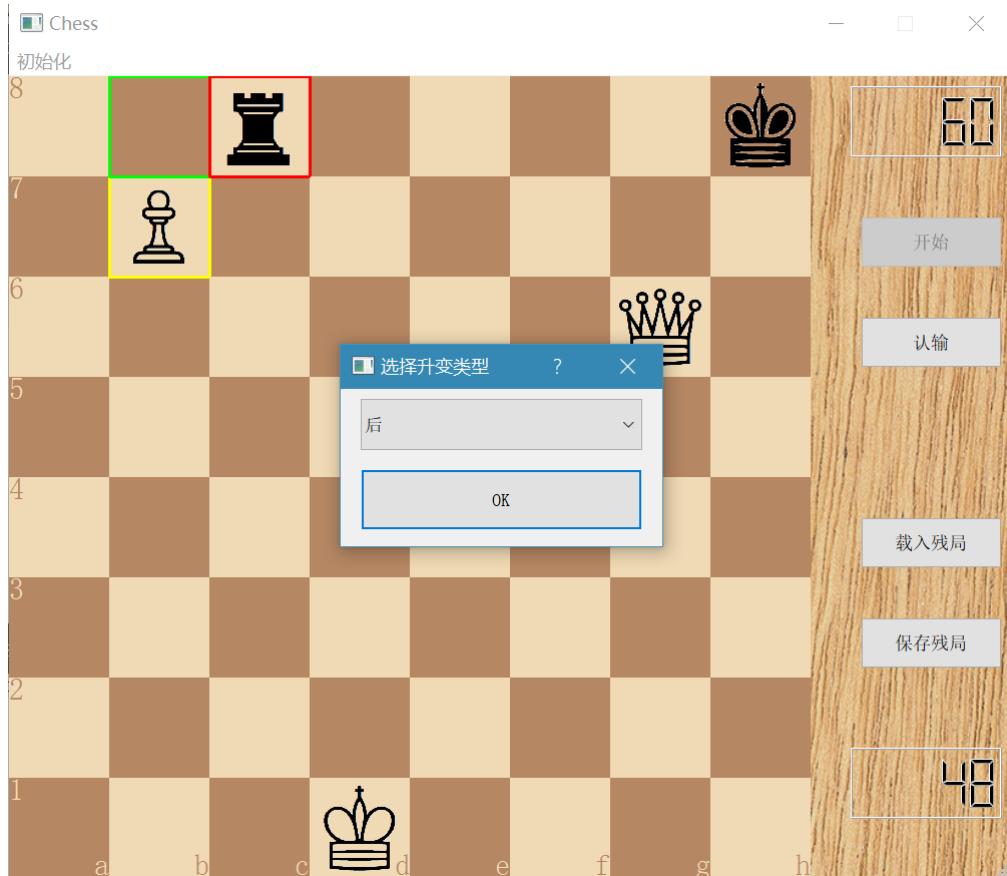
本程序的特色之一是客户端和服务端的不对称权限。一个不成文的规定是，数据的运算和主动权握在服务端手里，于是只有服务端能点击开始游戏，能够载入残局等。在服务端点击开始游戏，双方同时开始



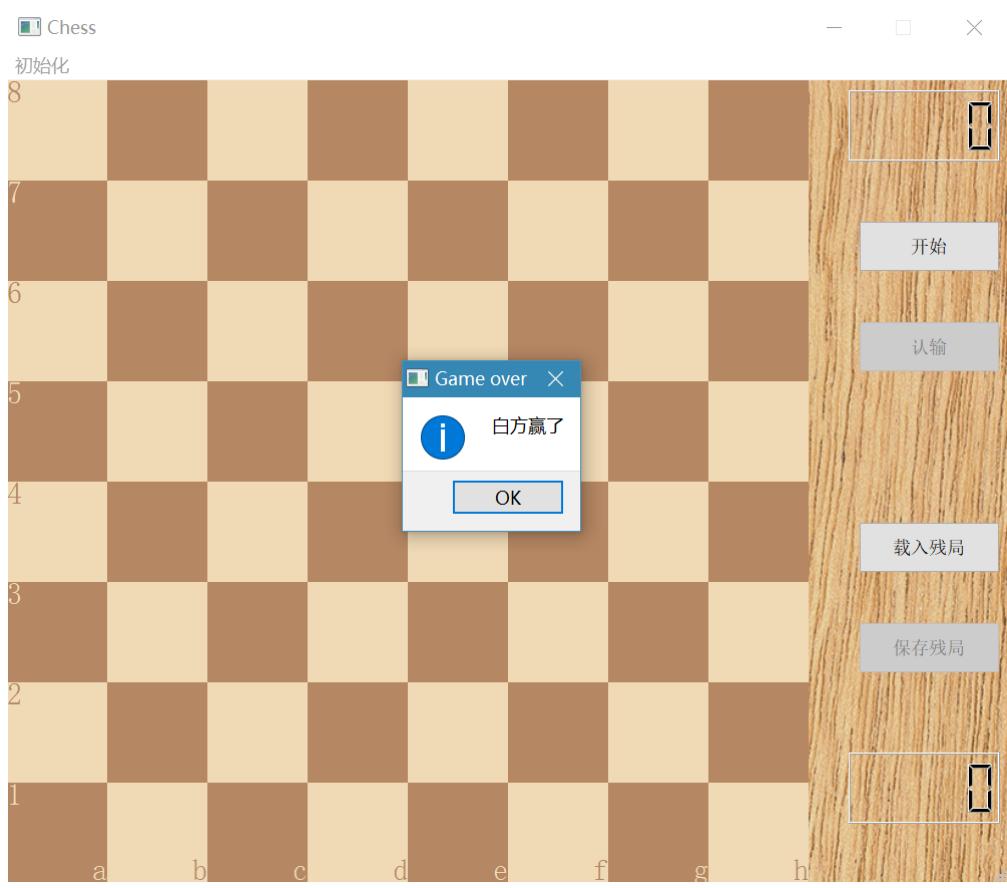
双方均可正常走子。此时黑方不走，倒计时结束后，双方均弹出消息框



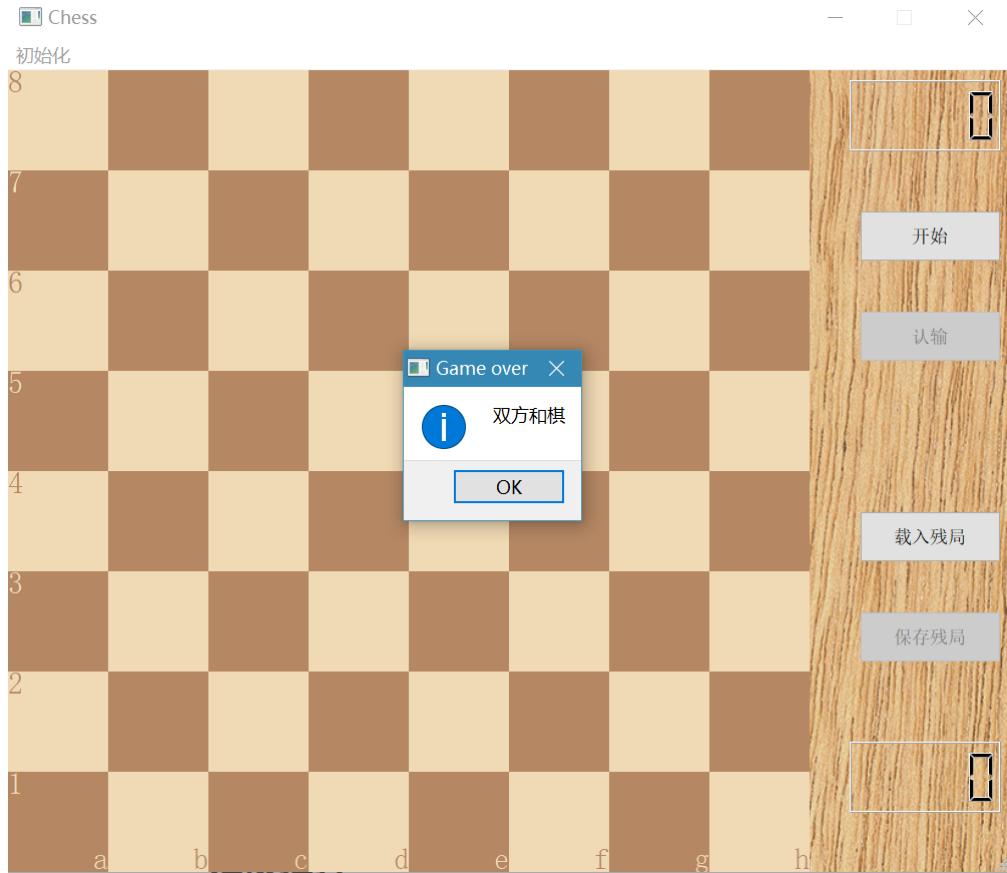
认输和将死效果相同，不再演示。载入兵升变残局，将兵走到底线，出现如下界面



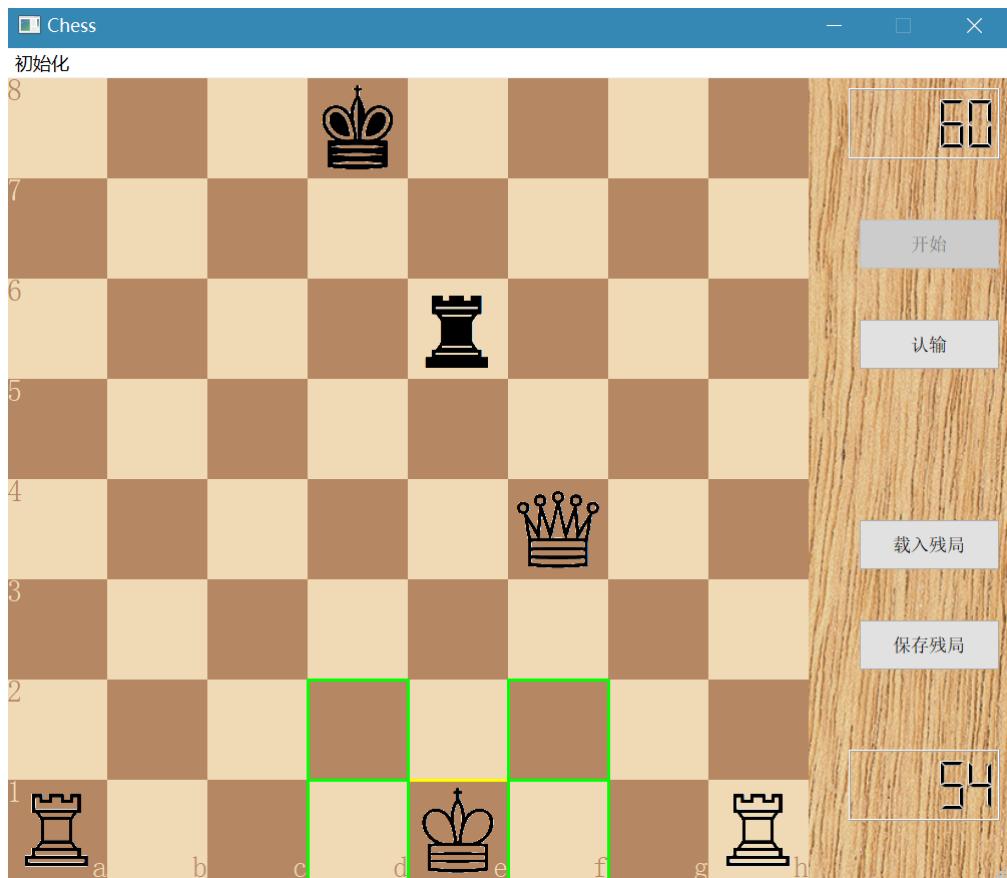
若选择后，则将死对方



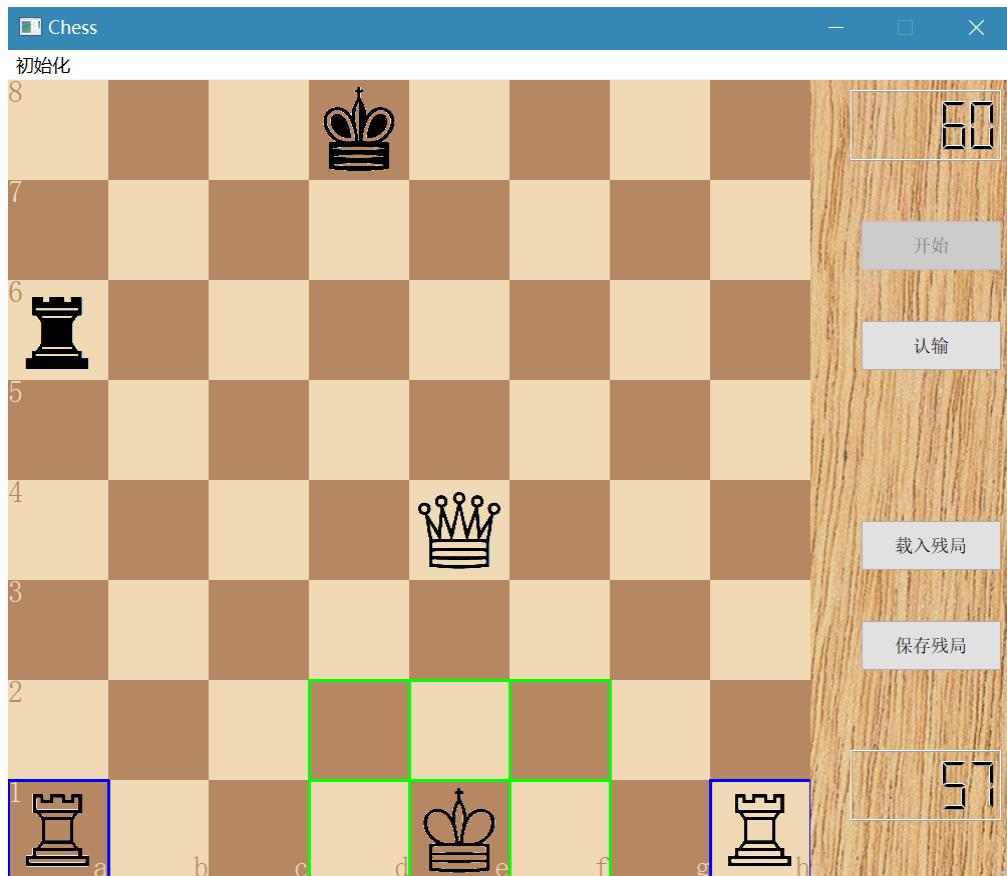
若选择象，则逼和



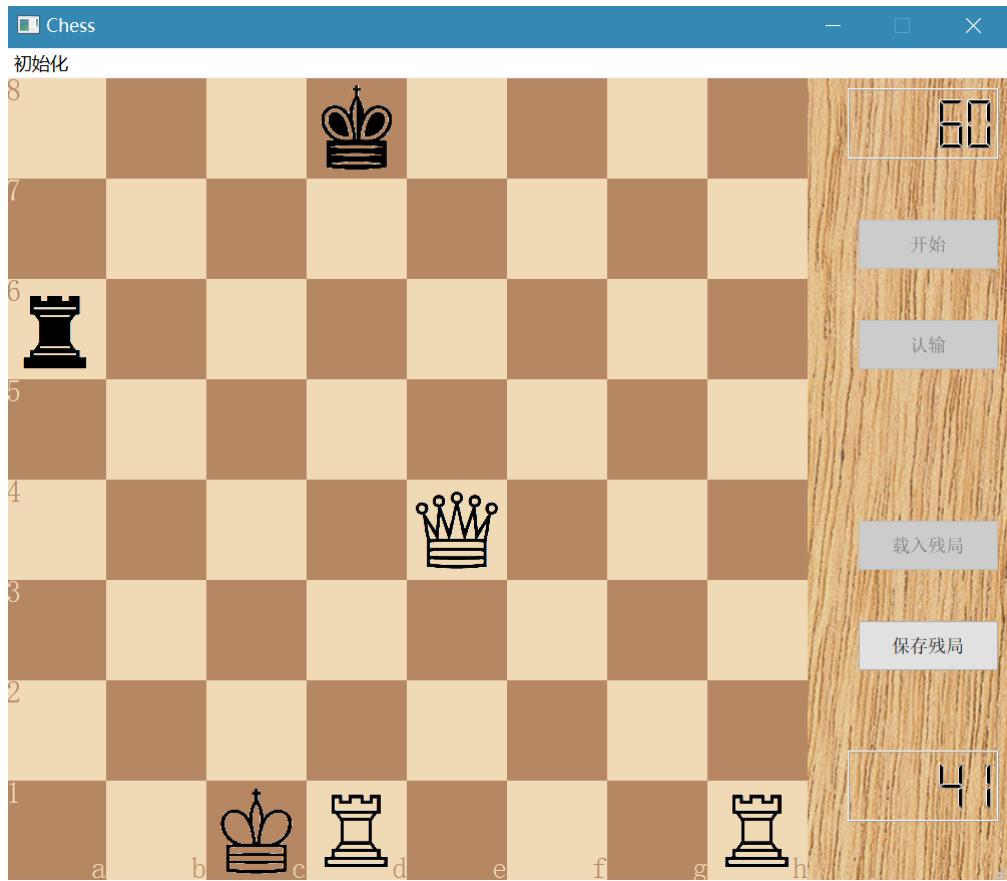
最后演示车马易位。载入残局



此时由于王受黑车威胁，无法进行车马易位



将局势走成上图所示，此时王以及王经过的路径均不受威胁，可进行长短易位。长易位后效果如下



## 程序设计

### 技术基础

本程序在Qt Creator中编写测试，共约1400行代码，使用自带编译器(mingW)编译。

使用了Qt自带的`QString`文本库，`QFile`文件库， `QMap`、 `QSet`、 `QVector`、 `QList`等容器库存储数据， `QTimer`进行计时， `QPaintEvent`、 `QMouseEvent`处理鼠标和绘图事件， `QMessageBox`、 `QFileDialog`标准消息框和文件对话框， `QPainter`绘图方法；没有使用 `QThread`进行多线程并发操作；网络通信部分使用的是Qt Socket部分的 `QTcpServer`和 `QTcpSocket`类，以及数据打包和处理时的 `QDataStream`， `QByteArray`等。

### 类的职责和关系

`utils.h`中首先定义了方便数据传输和运算的结构体包装，并重载了`QDataStream`的流运算符用于结构体的序列化、二进制化，编码和解码的过程

```

struct Pos{//坐标二元组
    int x,y;
    bool operator == (const Pos & value) const{//用于比较以及取得作qset键值的资格
        return (value.x==x&&value.y==y);
    }
    bool operator < (const Pos & value) const{//用于比较以及取得作QMap键的资格
        if(x<value.x) return true;
        if(x>value.x) return false;
        if(y<value.y) return true;
        if(y>value.y) return false;
        return false;
    }
    friend QDataStream& operator>>(QDataStream& in, Pos& s);
    friend QDataStream& operator<<(QDataStream& out, const Pos& s);
};
```

```

struct Piece{//棋子
    bool white;//棋子属于白方与否
    int type;//棋子类型, 0~5 queen bishop rook knight pawn king
    friend QDataStream& operator>>(QDataStream& in, Piece& s);
    friend QDataStream& operator<<(QDataStream& out, const Piece& s);
};

struct walk{//棋子的移动, 将王车易位也看做一种移动, 在下面的move()函数中一视同仁进行充分处理
    Pos pos;//目标位置
    int attack;//移动的附加属性 0:走 1:攻击 2:王车易位
};

struct Situation{//对局面的快照
    //单独标记王的位置
    Pos black_king;
    Pos white_king;
    QMap<Pos,Piece> pieces;//记录全局所有位置-棋子映射
    friend QDataStream& operator>>(QDataStream& in, Situation& s);
    friend QDataStream& operator<<(QDataStream& out, const Situation& s);
};

inline uint qHash(const Pos key){//用于结构体契合QSet的key, 与大作业一的功能相同, 不再赘述
    return key.x + key.y;
}

```

在命名空间 `utils` 中，封装了一系列算法相关的函数，是本程序的核心部分。由于程序禁止送吃行为，需要将死、逼和、王车易位功能，在得到棋子可走范围时需要进行虚拟的模拟，于是有了两个步骤的处理。这样的处理也使得将死和逼和的判断轻而易举：当某方所有子的FinalRange的并集为空集时，若此时此方被将军，则是将死；否则为逼和。

```

inline Situation getDefaultSituation();//硬编码, 得到初始局面
inline void move(Situation& s, Pos pos1, Pos pos2);//给定初始位置和目标位置, 在传入的局面上模拟移动(含王车易位的判断)

inline QVector<walk> getWalkRange(Situation& s, Pos pos);//给定棋子位置, 不禁止送吃, 得到最浅层的棋子移动范围

inline bool isKingVulnerable(Situation& s, bool white);//对某个局势判定某方的王是否受到威胁(在另一方棋子的攻击范围并集中)
inline QVector<walk> getFinalRange(Situation& s, Pos pos);//给定棋子位置, 得到最终的可移动范围
inline void save(QWidget *parent, Situation s, bool white);//给定当前方, 弹出对话框保存传入局势
inline bool load(QWidget *parent, Situation &tem, bool &white);//弹出对话框打开残局, 读取局势和当前方

```

`packet.h` 中将 `QTcpSocket` 传入或传出的数据流分拆或合并成独立的包的概念并提供信号和接口方便处理

```

public:
    explicit Packet(QObject *parent, QTcpSocket *socket);//传入用于包装的*socket
    void sendMsg(QByteArray data);//发送数据包
signals:
    void newPacket(QByteArray data);//收到数据包, 每个信号携带的都是单独的包

```

**updialog.h**中的 `UpDialog` 继承自 `QDialog`，是进行兵生变选择的对话框，调用其 `int getSetting()` 函数将执行 `exec()` 并阻塞，窗口销毁后返回选择的生变类型。

**connectiondialog.h**提供了一开始选择服务端/客户端并创建连接的窗口。

```
public:  
    explicit ConnectionDialog(QWidget *parent);  
signals:  
    void serverdone(QTcpSocket*);  
    void clientdone(QTcpSocket*);
```

父窗口只需新建 `ConnectionDialog` 对象进行初始化并进行信号的连接即可。当用户选择建立服务器并成功创建连接时触发携带socket的 `void serverdone(QTcpSocket*)` 信号，当用户选择建立客户端并成功创建连接时触发携带socket的 `void clientdone(QTcpSocket*)` 信号。

**chessboard.h**是国际象棋棋盘的显示控件，内部将处理鼠标的点击事件、可移动区域的计算和显示等。对外层来说重要的只有

```
public:  
    explicit ChessBoard(QWidget *parent);  
    void setSituation(Situation s); //设置局面，立刻repaint()生效  
    void setStatus(int id); //设置棋盘显示状态 //0:empty 1:loaded 2:selecting  
    //下面两个函数控制着某棋子能否被点击等  
    void setSide(bool iw); //设置本端是白方还是黑方  
    bool isnowwhite; //设置当前是哪一方走棋  
signals:  
    void move(Pos pos, Pos pos2); //当用户点击鼠标进行棋子的移动时触发信号，携带棋子位置和目标位置
```

**mainwindow.h**包含了主窗口 `Mainwindow`。在本次设计中主窗口负责的功能较少，它主要是起到调整界面的作用。它在一开始创建了 `ChessBoard*`，但当客户端或服务端创建完成时，便对应创建了 `chessclient.h` 中的 `chessClient` 或 `chessserver.h` 中的 `chessServer`，并将 `chessboard*` 交予管理。可以认为 `chessClient` 和 `chessServer` 起到的是 proxy 的作用，它们负责客户端和服务端间的网络通信，在一端构成了 `ChessBoard`↔`Proxy`↔`MainWindow` 的沟通模式。

### 客户端、服务端工作流程

下面列出了客户端/服务端单方面 `ChessBoard`↔`Proxy`↔`MainWindow` 可能出现的沟通类型。

proxy到mainwindow的通信  
开始(c/s)  
换某边(c/s)  
结束(c/s)  
计时器更新(c/s)

mainwindow到proxy的通信  
开始(s) 未开始  
认输(c/s) 此方  
载入残局(s) 此方  
保存残局(c/s) 此方

board到proxy的通信  
移动(c/s)

mainwindow到board的通信  
设置状态(c/s)

设置本端是黑/白方(c/s)

proxy到board的通信  
设置局面(c/s)  
设置状态(c/s)  
设置当前方(c/s)

在程序中，这些可能的情况均使用信号槽或者公共方法来进行，不再详细叙述。mainwindow主要负责主界面的调整（按钮的disable等）， proxy负责网络通信和模块间通信， board则进行棋盘的更新显示。

### 网络通信编程框架

这里着重介绍 QTcpSocket 的封装类 Packet。Packet 头文件的完整定义为

```
class Packet : public QObject
{
    Q_OBJECT
public:
    explicit Packet(QObject *parent, QTcpSocket *socket);
    void sendMsg(QByteArray data);
signals:
    void newPacket(QByteArray data);

private slots:
    void received();
private:
    QTcpSocket *socket;
    uint next_block_size=0;
};
```

其构造函数除了存储\*socket之外，还执行了

```
connect(socket, SIGNAL(readyRead()), this, SLOT(received()));
```

即托管了socket\*的数据接收，处理后再以 void newPacket(QByteArray data) 的signal发送出去。Packet 的出现是为了规避TCP/IP协议可能出现的粘包/拆包问题（这在此程序中是必要的，由于此程序设计的思想所致，客户端和服务端的通信来回次数较多，程序中具有连续发送两个包的情况，因此必须处理拆包）。发包要通过 void sendMsg(QByteArray data) 函数进行，它的内容为

```
void Packet::sendMsg(QByteArray data){
    QByteArray rec;
    QDataStream sendStream(&rec, QIODevice::ReadWrite);
    sendStream << uint(data.size()) << data;

    socket->write(rec);
}
```

可以看到处理方式是在数据的头部加入数据的长度信息，再发送出去。而对接收到信息的处理为

```
void Packet::received(){
    QDataStream clientReadStream(socket);
    while(true) {
        if (!next_block_size) {
            if (socket->bytesAvailable() < sizeof(uint)) {
```

```

        break;
    }
    clientReadStream >> next_block_size;
}
if (socket->bytesAvailable() < next_block_size) {
    break;
}
char* raw;

clientReadStream.readBytes(raw,next_block_size);
QByteArray data(raw,next_block_size);
emit newPacket(data);
next_block_size=0;
}
}

```

当接收数据的长度不足以读取头部的长度信息，或接收的数据量小于应接收的数据长度时，程序将等待下一次接收，这样就解决了拆包。而数据量足够时，程序读取头部的数据长度信息并读取对应长度的数据，这样便完成了一个独立的数据包的提取；而每成功提取一个包，就发送携带它的信号，这样就解决了粘包。经过这一个中间层的包装，数据的收发轻易很多，如在chessclient中，开始时进行信号连接

```
connect(packet,&Packet::newPacket,this,&ChessClient::READ);
```

READ(QByteArray) 接收的便是完整独立的包，可以直接进行解析。

```

void ChessClient::READ(QByteArray data){
    QDataStream in(data);
    int type;
    in>>type;
    if(type==0){
        ...
    }
}

```

发包时，可以简单地调用 sendMsg(QByteArray data) 函数，例如

```

void ChessClient::move(Pos pos1,Pos pos2){
    QByteArray data;
    QDataStream out(&data,QIODevice::ReadWrite);
    out<<0<<pos1<<pos2;
    packet->sendMsg(data);
}

```

## 通信协议

之前提到过，本程序采用不对称通信设计。这个构想的初衷是实现一定程度的反作弊功能，这个框架下若期望拓展到游戏平台的构建（而非双人对战），修改会减少很多。对于游戏平台来说，要不惮以最坏的恶意来揣测玩家。因此应该将一切的计算、控制权限交予服务端，否则，举例说客户端采用强行修改内存等方法像服务端发出了非法、不合理动作（如开局直接用兵将对方王吃掉），服务端若想避免惨剧只能加入繁琐的检测机制（实际上很多即时性的网游就是这样干的，不过这只是因为射击游戏等对实时性的刚需，因为网络延迟的原因全部交予服务器处理数据不成立）。而国际象棋这样的对战不需要很高的即时性，所以完全可以采取客户端只向服务端发送可行的动作，服务端处理局面后将结果返回客户端（同时也更新服务端的界面和变量等），客户端进行显示的方法。

下面是客户端和服务端的通信协议。

```
//server to client //0:发送局面 //1:发送双方计时 //2:发送胜/负/和 0胜1负2和 //3:换为X边 0白1黑 //4:  
请求兵升变 0~3 queen bishop rook knight //5.开始 白方先走 只需调整变量 紧接着还要发换边信息  
//client to server //0:走棋 //1:升变 //3:认输
```

从上一个部分的代码中可以看到，每个数据包的头部都是一个整数，代表的就是信息类型；之后是具体数据。对于客户端的动作，客户端向服务端发送请求，服务端处理后更新自身界面、变量并发送使客户端更新界面、变量的数据包，客户端接收后进行更新。计时的变动、胜负和的消息、换边的消息、改变局面棋子排布、弹出生变对话框、开始一局的动作都是服务端发起，客户端只能接收并显示。