案卷号	00001
日期	

软件详细设计说明书（例）

作　　者：_____

完成日期：_____

签 收 人：_____

签收日期：_____

修改情况记录：

版本号	修改批准人	修改人	安装日期	签收人

目录

1 引言

1.1 编写目的

随着证券交易电子化程度的不断提高，券商对于各种业务提出了新的要求，为了满足券商的发展需求，更好的为客户提供服务，现结合原有各版本的证券交易软件的优点和特点，开发一套采用 Client/Server 结构的证券交易软件管理系统（ SQL 版）。本系统从底层予以优化，使整个系统的运行速度得到较大提高，通过重新优化数据库内部结构，使系统的可扩充性得到极大提高。

本说明书给出 SQL 版证券交易系统的设计说明，包括最终实现的软件必须满足的功能、性能、接口和用户界面、附属工具程序的功能以及设计约束等。

目的在于：

为编码人员提供依据；

为修改、维护提供条件；

项目负责人将按计划书的要求布置和控制开发工作全过程；

项目质量保证组将按此计划书做阶段性和总结性的质量验证和确认。

本说明书的预期读者包括：

项目开发人员，特别是编码人员；

软件维护人员；

技术管理人员；

执行软件质量保证计划的专门人员；

参与本项目开发进程各阶段验证、确认以及负责为最后项目验收、鉴定提供相应报告的有关人员。

合作各方有关部门的复杂人；项目负责人和全体参加人员。

1.2 范围

说明：

a. 待开发的软件系统的名称： 模拟股票交易系统

b. 列出本项目的任务提出者、开发者、用户以及将运行该项软件的单位。

1.3 定义

列出本文件中用到的专门术语的定义和缩写词的原词组。

本报告用到的术语符合国家标准《软件工程术语（ GB/T11475-1995 ）》。

1.4 参考资料

列出要用到的参考资料，如：

a. 本项目的经核准的计划任务书或合同、上级机关的批文；

b. 属于本项目的其他已发表的文件；

c. 本文件中各处引用的文件、资料，包括所要用到的软件开发标准。

列出这些文件的标题、 文件编号、 发表日期和出版单位， 说明能够得到这些文件资料的来源。

2 总体设计

2.1 需求规定

说明对本系统的主要的输入输出项目、 处理的功能性能要求, 详细的说明可参见 《需求分析说明书》 。

2.2 运行环境

简要地说明对本系统的运行环境 （包括硬件环境和支持环境） 的规定, 详细说明参见 《需求分析说明书》 。

数据库服务器

奔腾 Pro

内存 128MB以上

硬盘 9GB

100M 网卡

应用服务器

奔腾 Pro

内存 64MB以上

硬盘 4GB

100M 网卡

网络配置

100M / 10M

工作站 (柜台)

P100 以上

内存 8MB以上

硬盘 1G 以上

100M/10M网卡

软件

操作系统

Windows NT 4.0 以上

数据库管理系统

SQL Server 2005

相关软件工具

Windows NT Workstation/Windows NT server

Windows 2000 Professional/ Server

开发工具

平台： Windows95/98 、 Windows NT、 Windows 2000

开发工具： visual stidio 2005 sp1,C#.Net

测试环境

Windows31 、Windows95/98 、 Windows NT、 Windows 2000

2.3 基本设计概念和处理流程

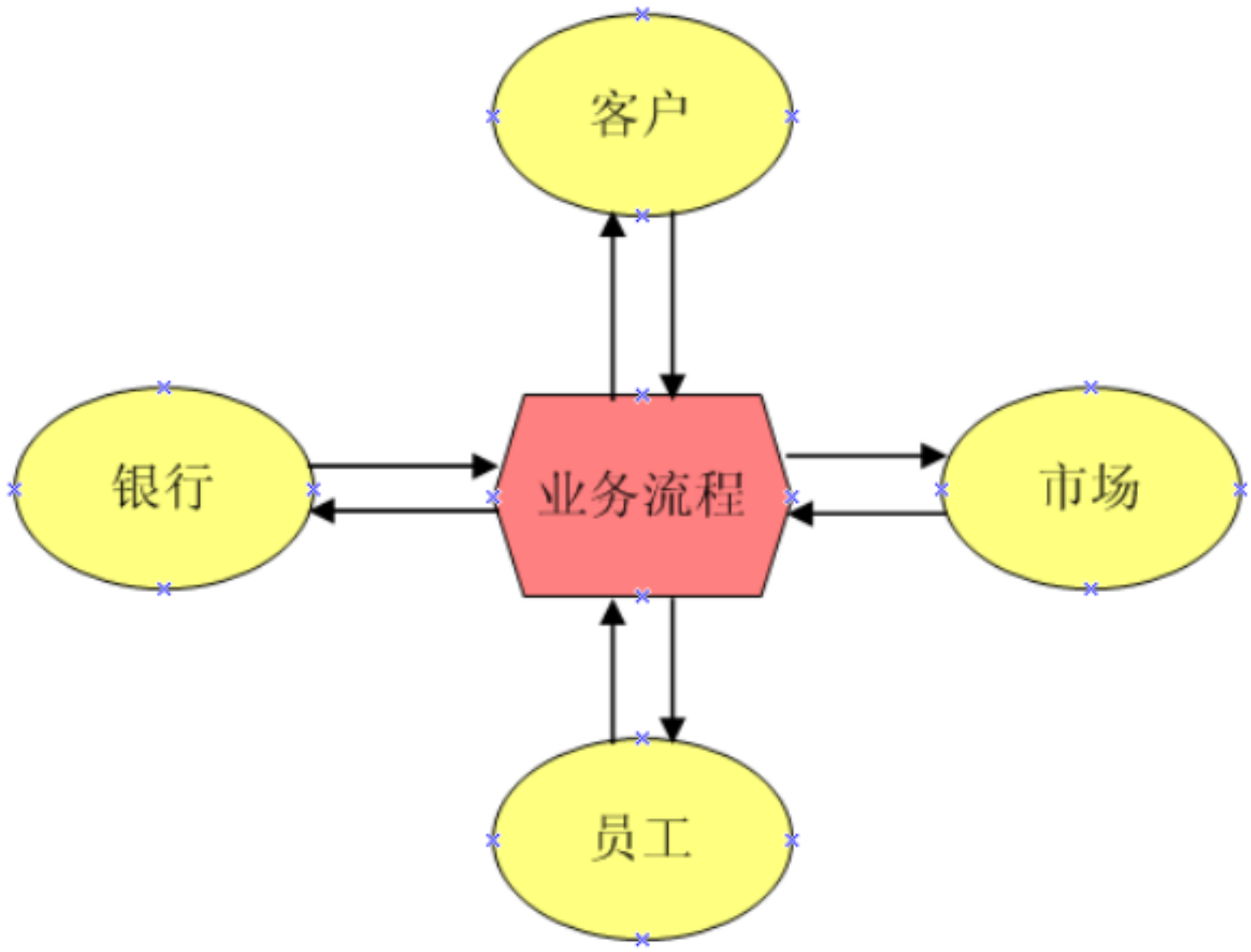
说明本系统的基本设计概念和处理流程，尽量使用图表的形式。

营业部系统一共有四个对象， 即客户、 员工、 市场和银行， 市场的概念是交易所的细化，比如上海证券交易所的A股和B股就是两个市场， 有了市场的概念我们就可以把交易所这个概念细化， 并使同一个市场的共性更突出。 银行则通过银证转账业务介入， 并成为营业部系统不可或缺的组成部分。

上述四个对象通过一些业务流程进行相互操作从而形成整个交易活动。 因此整个系统模

型可以表述为图　2-1

设计时需要将营业部系统所使用的各种信息分为描述四个对象的信息和描述业务流程的信息。 由于四个对象相对而言是一种稳定型信息，　而业务流程则较易变化，　且营业部之间差异很大，　因此应将四个对象尽量定型，　而将各种业务流程尽可能做成组件，　以便营业部可根据实际需求组装成适合自己的系统。



根据以上思想，在设计对象模型时应充分考虑到可扩展性，尽量做到抽象化、参数化，从而使对象需求变化时不致影响系统结构。

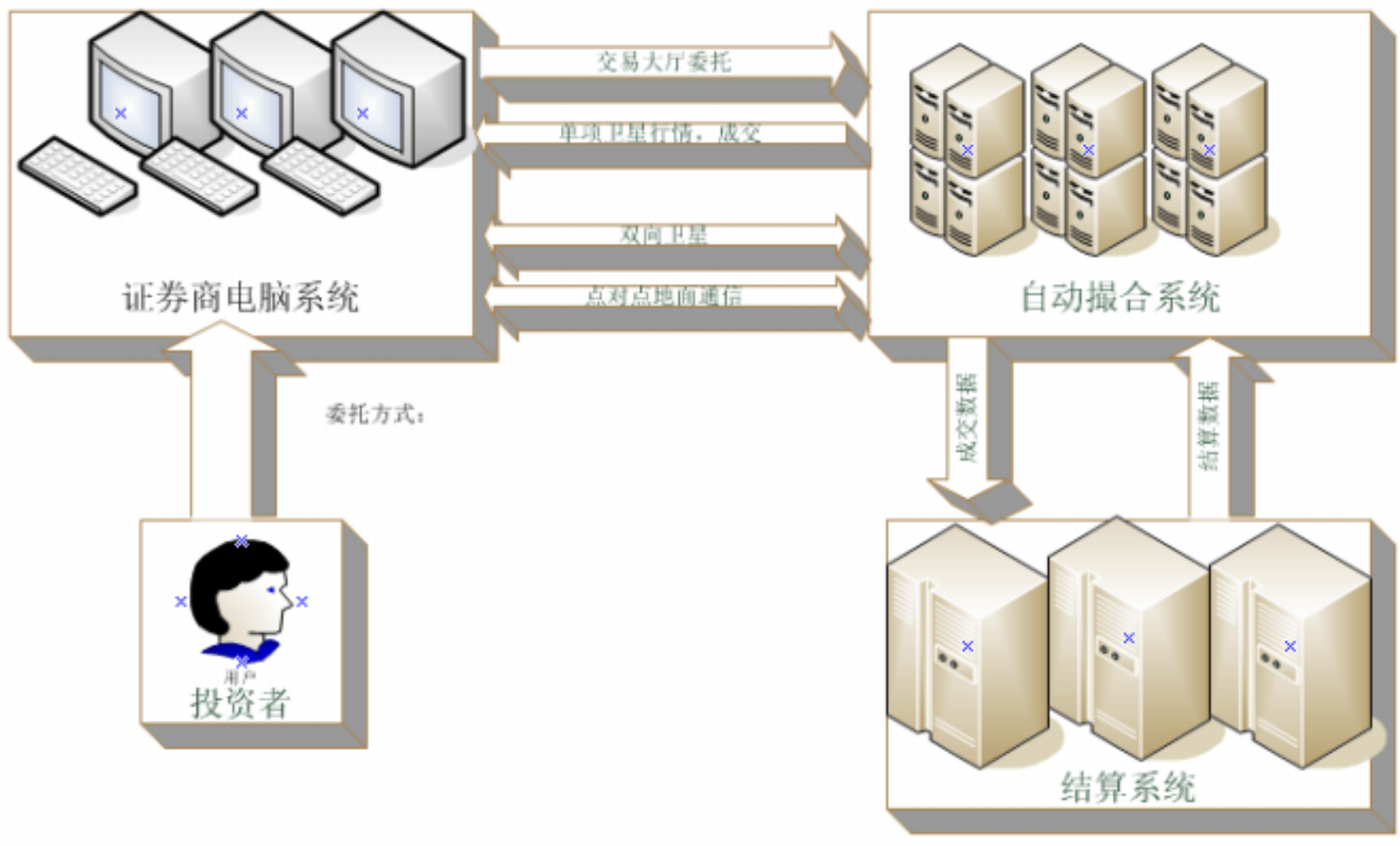


图 2.1

2.4 结构

用一览表及框图的形式说明本系统的系统元素 （各层模块、子程序、公用程序等） 的划分，扼要说明每个系统元素的标识符和功能， 分层次地给出各元素之间的控制与被控制关系。

本系统采用 c/s 模式的 3 层结构

按照不同会话来划分的话可以分为 3 大系统模块

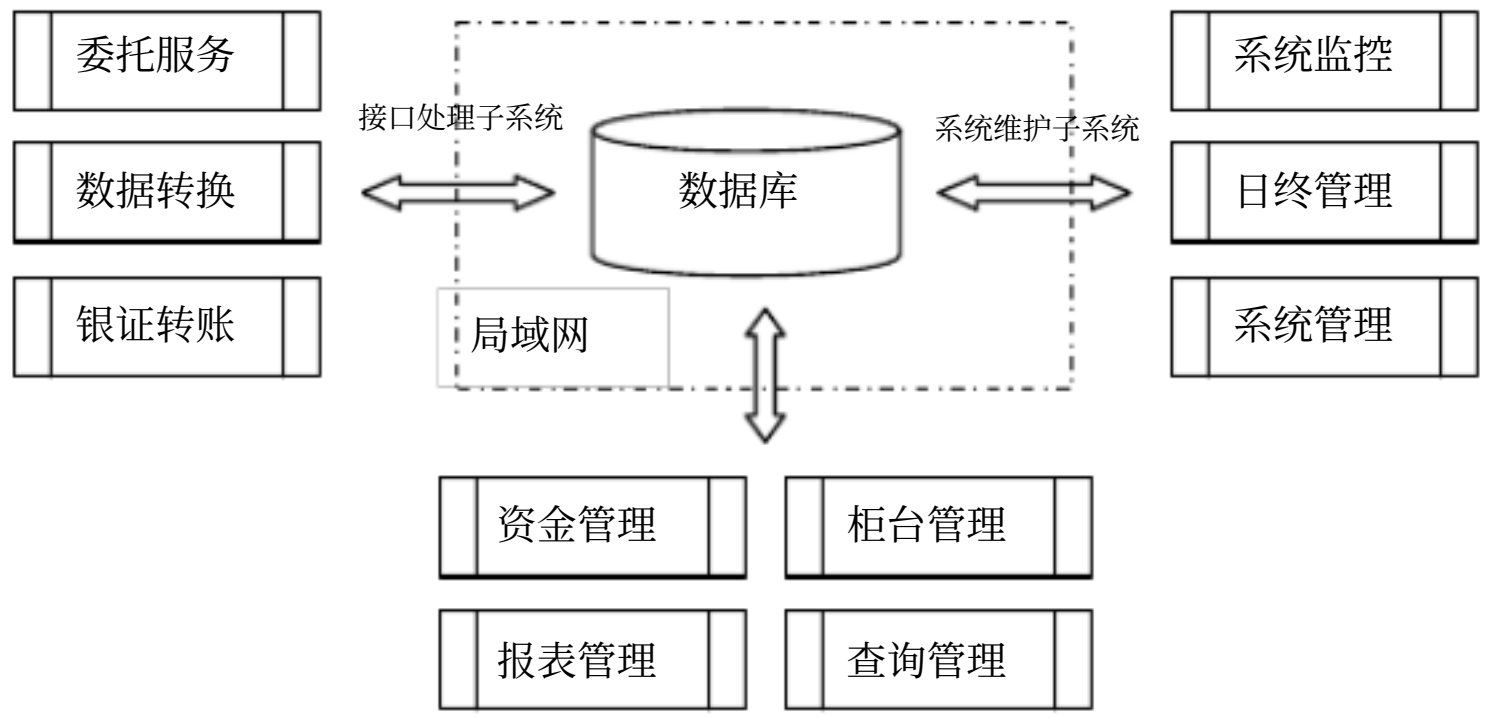
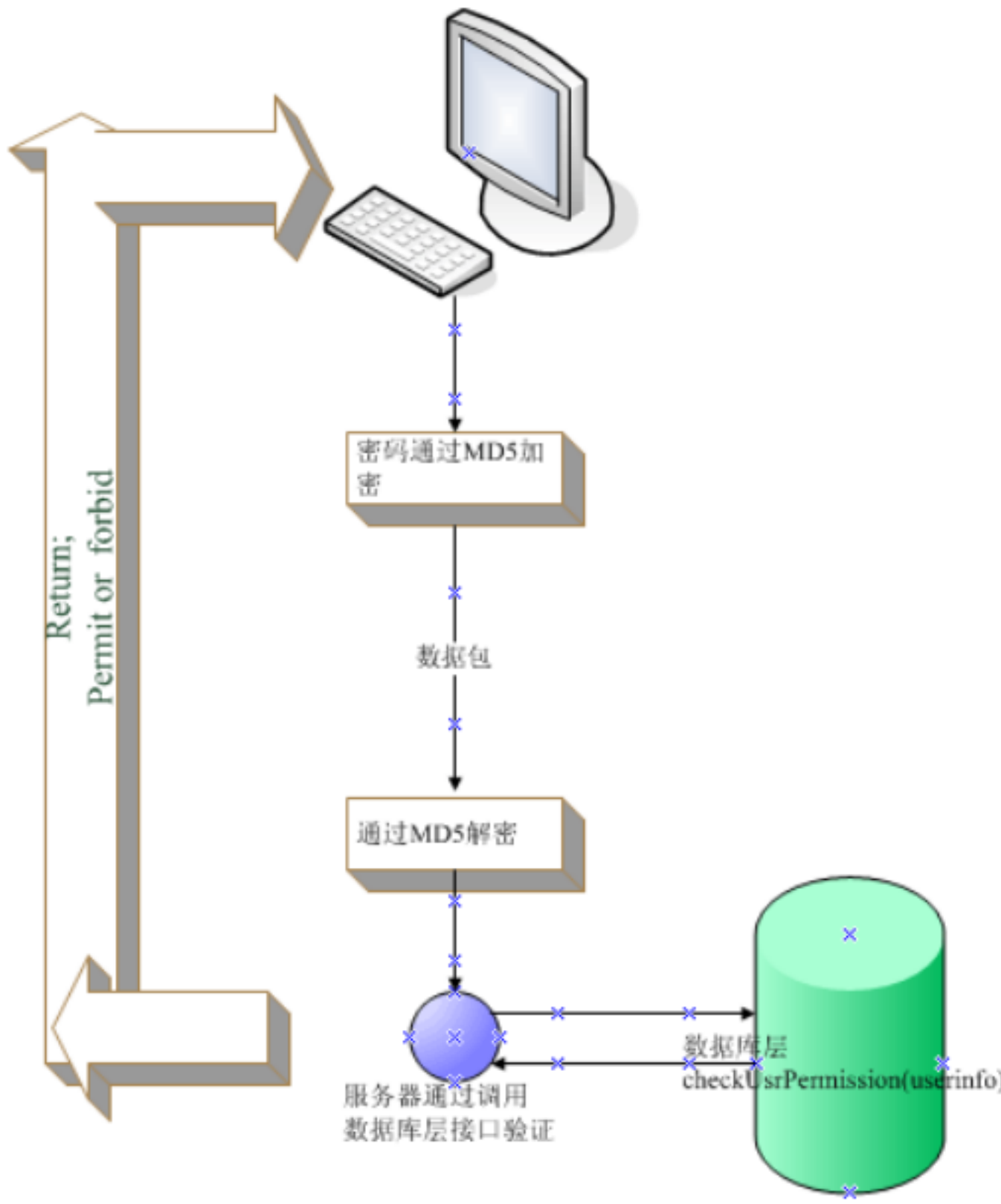
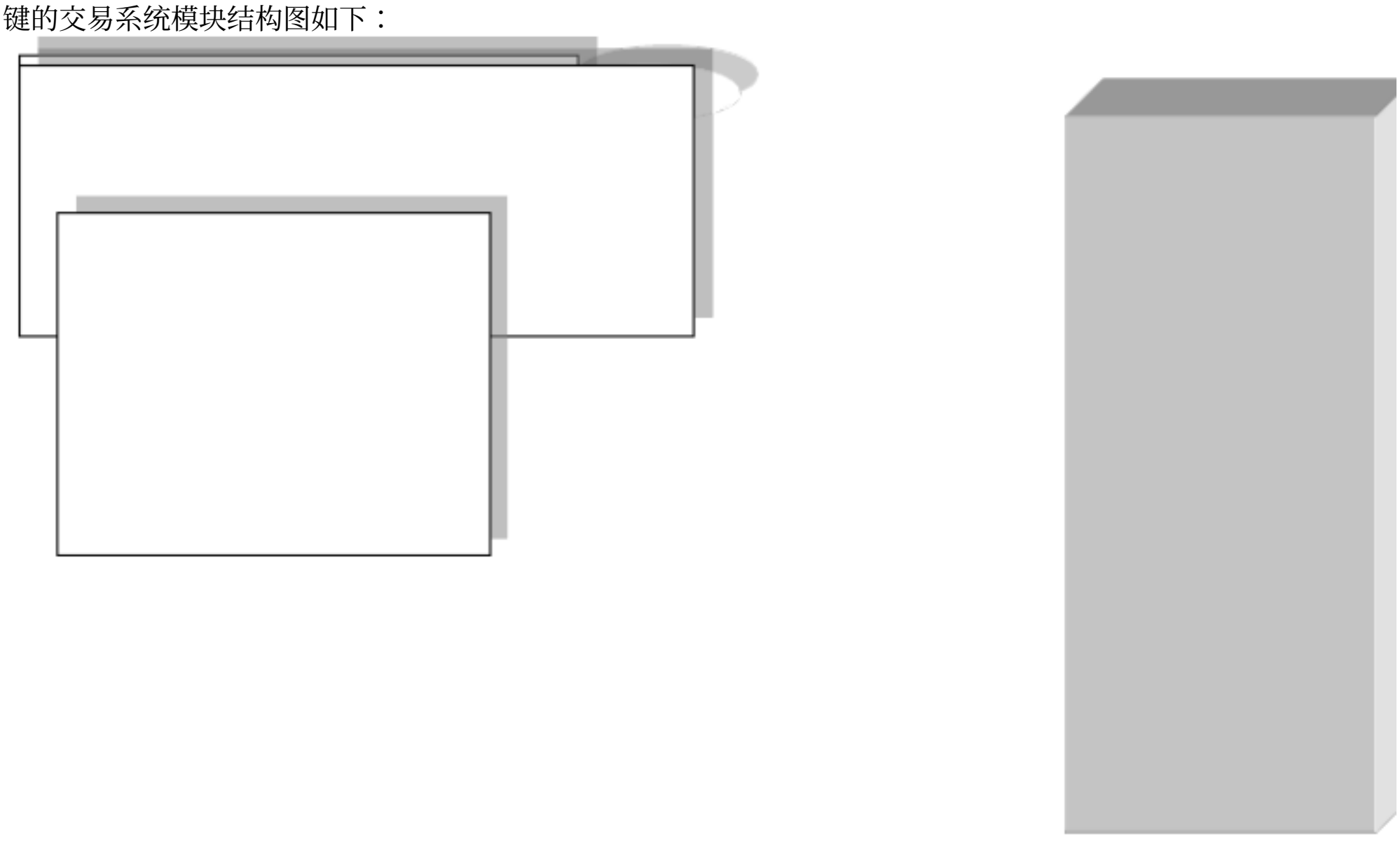


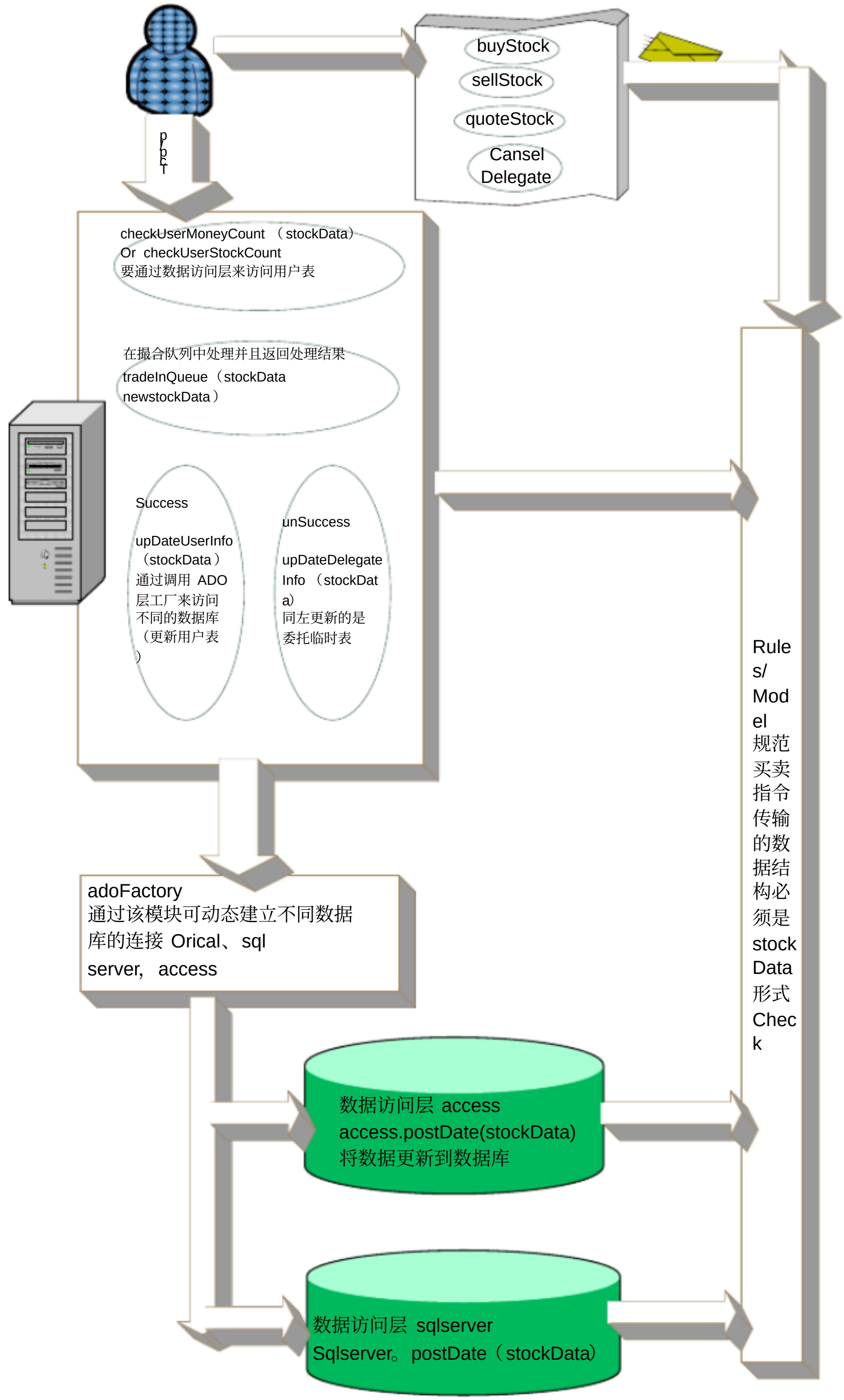
图 2-2 交易系统体系结构

客户端登陆模块：



最关键的交易系统模块结构图如下：

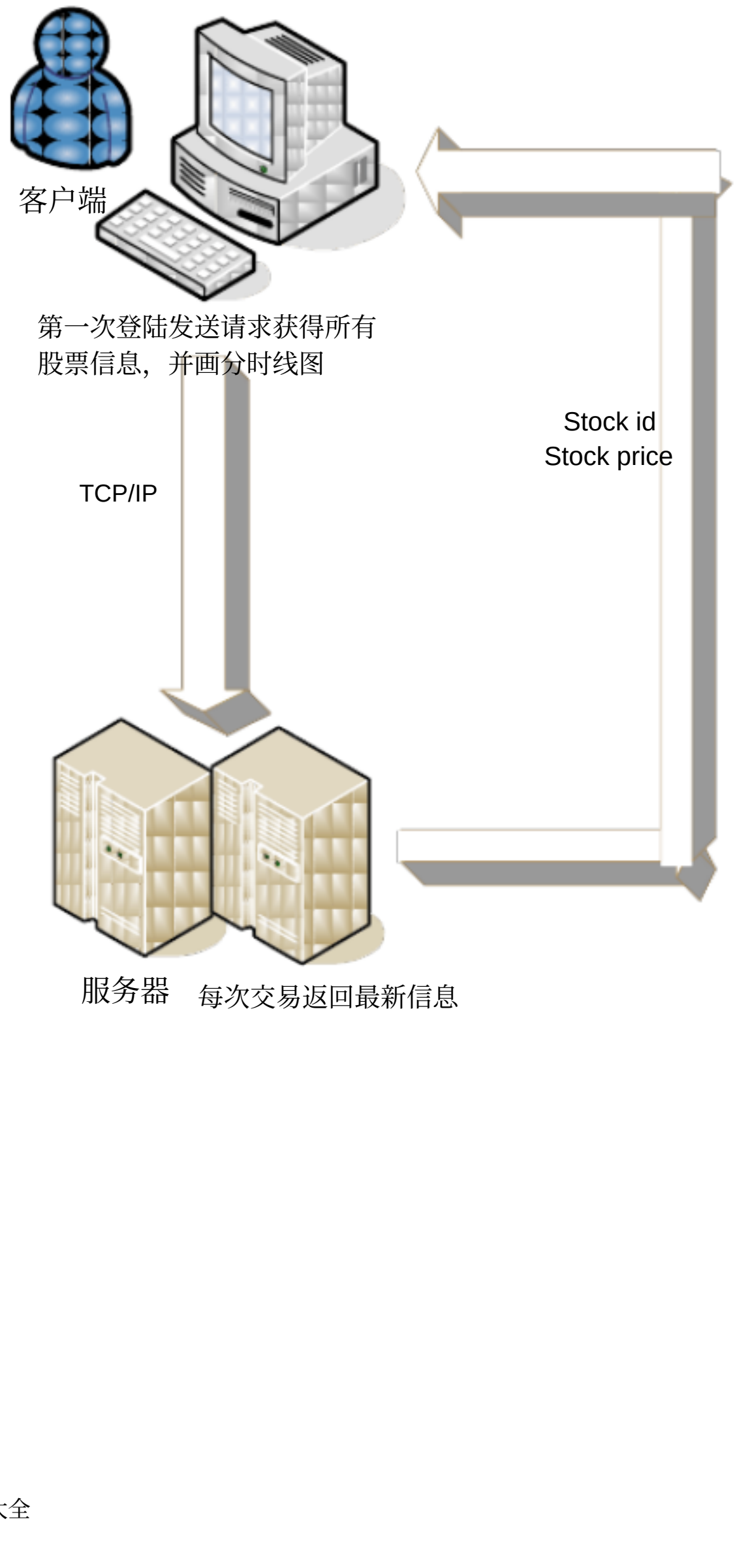



buyStock
sellStock
quoteStock
Cansel
Delegate
post
checkUserMoneyCount （stockData）
Or checkUserStockCount
要通过数据访问层来访问用户表
在撮合队列中处理并且返回处理结果
tradeInQueue（stockData
newstockData）
Success
upDateUserInfo
（stockData）
通过调用 ADO
层工厂来访问
不同的数据库
（更新用户表
）
unSuccess
upDateDelegate
Info （stockDat
a）
同左更新的是
委托临时表
Rule
s/
Mod
el
规范
买卖
指令
传输
的数
据结
构必
须是
stock
Data
形式
Chec
k
adoFactory
通过该模块可动态建立不同数据
库的连接 Orical、 sql
server， access
数据访问层 access
access.postDate(stockData)
将数据更新到数据库
数据访问层 sqlserver
Sqlserver。 postDate（stockData）

股票信息发布

经过修改我认为每次由客户端每 5 秒去查询一次服务器更新信息不可取，因为这会加重服务端和客户端的负担，特别是服务器端的运算。

修改后实现变更为： 用户一开始登陆后获得一次服务器的全部股票当前信息。而服务器端每次发生交易后， 给每一个在线用户发送当前交易需要更新的股票信息，这样就减轻了客户机和服务端的信息



2.5 功能需求与程序的关系

（该关系由需求分析报告编写者根据结构图说明）

本条用一张如下的矩阵图说明各项功能需求的实现同各块程序的分配关系：

	获取并发送用户请求	绘制分时图	MD5 加密解密	发送用户交易请求	接受并识别用户请求	调用数据层查询	撮合交易	服务器返回客户端信息
用户登陆	√		√		√	√		
查看用户持仓	√				√	√		
实时指数	√	√						√
交易委托	√		√	√	√	√	√	√
取消交易	√		√	√	√	√		√

2.6 人工处理过程

说明在本软件系统的工作过程中不得不包含的人工处理过程（如果有的话）　　。

没有完成股票管理的模块设计，所以股票必须从数据库后台添加

如果有新股发行，还必须添加有关股票的交易队列

2.7 尚未解决的问题

说明在概要设计过程中尚未解决而设计者认为在系统完成之前必须解决的各个问题。

3 接口设计

3.1 用户接口

说明将向用户提供的命令和它们的语法结构，以及软件的回答信息。

向用户提供简单易用的　　UI，以及帮助文档。

客户端将提供以下功能

首先弹出用户登陆框，供用户输入用户名和密码

菜单项提供个股查询和分时图按钮

菜单栏下是选项卡，提供股票实时信息和个股分时图栏

提供用户交易界面和交易按钮以及查看用户盈亏按键

3.2 外部接口

说明本系统同外界的所有接口的安排包括软件与硬件之间的接口、　　本系统与各支持软件之间的接口关系。

采用基于正确公开标准的部件和技术以确保最大限度的协作能力以及与第三方系统与部件集成的简便性。这类标准包括但不限于以下几种：

网络协议与标准　(TCP/IP, HTTP, SSL, etc)

语言 (SQL, C#.net, etc.)

数据库连接性　(ADO。 net)

3.3 内部接口

说明本系统之内的各个系统元素之间的接口的安排。

逻辑层和数据访问层通过以经的　　stockDataModel 接口，来限定访问　stockData 类型的数

据

客户端通过调用 buyStock（stockData）和 sellStock（stockData）来访问逻辑层，在这个函数中包含了访问逻辑层的接口 dealTransaction(stockData)

通过 AdoFactory 访问不同的数据库

客户端登陆协议

D(二字节)+(客户名字长度)(4 字节)+(客户名字)+(客户密码长度)(4 字节)+(客户密码);

客户买卖协议

B(二字节)+(股票 ID)(4 字节)+(股票数量)(4 字节)

S(二字节)+(股票 ID)(4 字节)+(股票数量)(4 字节)

查询交易信息并返回给客户端

C(二字节)

具体有拆包解包的类

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ProjectCenterTradingSys
{
    public class Protocal
  {
        private byte [] messagebuffer;
        private byte [] messagelength;
        public byte [] messagebag;



      // 该函数是将字符串转换为字节数组
        public byte [] StringtoByte( string stringInfo)
    {
```

```
    messagebuffer = System.Text.              ASCIIEncoding  .ASCII.GetBytes(stringInfo);
            return    messagebuffer;
}



    // 该函数将整型转换为个字节
    public     byte [] InttoByte(       int    number)
{
    messagelength=          BitConverter    .GetBytes(number);


            return    messagelength;
}



    // 将浮点型转换为个字节
    public     byte [] DoubletoByte(      double  price)
{
            byte [] pricebyte =         BitConverter    .GetBytes(price);
            return    pricebyte;
}



    // 合并一个字符串（字节数组）和他的长度作为一个包
    public     byte [] Combinarray(     byte [] messle,       byte [] messinfo)
{
    messagebag=          new byte [messle.Length+messinfo.Length];

            int    index;
            for   (index = 0; index < messle.Length; index++)
messagebag[index] = messagelength[index];
            for   (  int    index1 = 0; index1 < messinfo.Length; index1++)
messagebag[index + index1] = messagebuffer[index1];
            return    messagebag;
```

```
}
        // 解包头
        public     byte [] BagHead(   char  head)
{
                byte [] headbyte =      BitConverter    .GetBytes(head);
                return    headbyte;
}
```

```
        // 读包头
        public     char  DeBagHead(byte [] buffer)
{
                char  headinfo =      BitConverter    .ToChar(buffer, 0);
                return    headinfo;
}
```

```
        // 该函数为解包信息为字符串     !
            public     string    deMessgeBag( byte [] Messagebag,    int   start,     out   int    next)
        {
           next =               BitConverter    .ToInt32(Messagebag, start);
                        string    message = System.Text.     ASCIIEncoding  .ASCII.GetString(Messagebag, start + 4,
next);

                        return    message;
        }
```

4 运行设计

4.1 运行模块组合

说明对系统施加不同的外界运行控制时所引起的各种不同的运行模块组合，说明每种运行所历经的内部模块和支持软件。

4.2 运行控制

说明每一种外界的运行控制的方式方法和操作步骤。

4.3 运行时间

说明每种运行模块组合将占用各种资源的时间。

5 系统数据结构设计

5.1 逻辑结构设计要点

给出本系统内所使用的每个数据结构的名称、标识符以及它们之中每个数据项、记录、文卷和系的标识、定义、长度及它们之间的层次的或表格的相互关系。

客户端类图：

windowForm ：Form

Private：

userLogDialog
userNametextBox
userPasswordtextBox
userlogOKbotton
userlogCanselbutton
tabPage
MenuBar
stockRealtimeGraphitem
stock Quote Dialog
dataGridView
userBuyStockID
userBuyStockcount
userBuyStockprice
userBuyStockButton
...sell
userStocklistView
userStockLookButton

---

send Mesto Server(string Info)
//该函数用来向主机发送请求
协议 U：发送用户名，密码
B：buy 股票 id,count,price,user
S: sell

(接上）

MD5encrypt （string）
//以下都要通过　sendMestoServer
//向主机发送信息
logOK_press(event,handle);

stockQuoteitem_press(e,h);

buyStockButton_press(e,h);

sellStockButton_press(e,h);

stocklookButton_press(e,h);

//////
该函数调用　drawPicture 画图
stockRealtimeGraphitem_press(e,h
)

Class RealTime Graph

Private

stockID

//动态数组存储股票价格

ArrayList stockPrice[]

Public ：

//在 windowform 类中 recievemess 后更新当前价格，即在数组后添加一项最新价格

updatePrice（price， sotckPrice)

drawPicture （ stockID ， stockPrice）

Class stockData

订单号 public int ListID;

public int UsrID;

public string StockIndex;

public flout Price;

public int Count;

public bool Isbuy;

该类即为向服务端传送数据时的包

服务器端

StockQueue

Private

stockData data

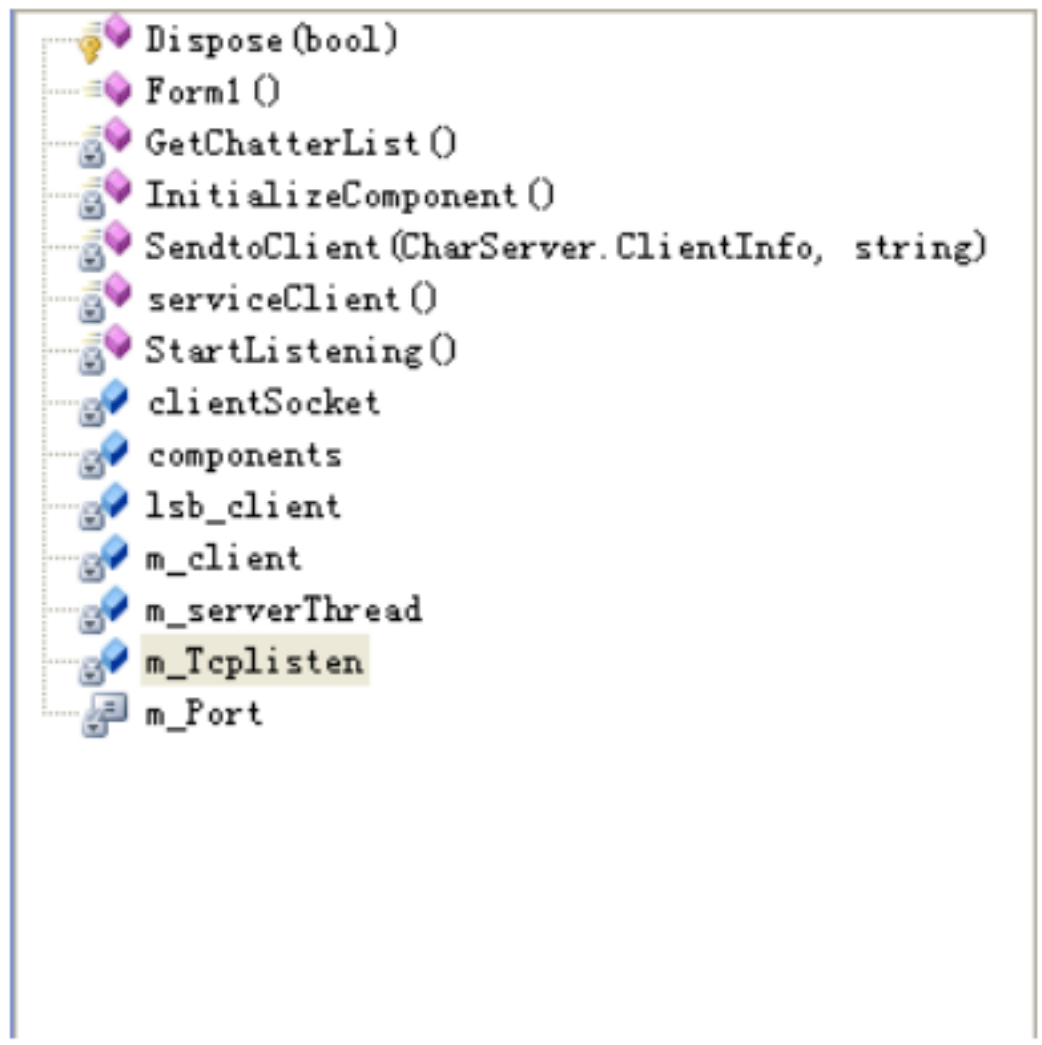
stockData next

Public

DeleteQueueHead();

AddStockData();

Class TradeService



该类还要补充若干个 StockQueue 类型的成员变量

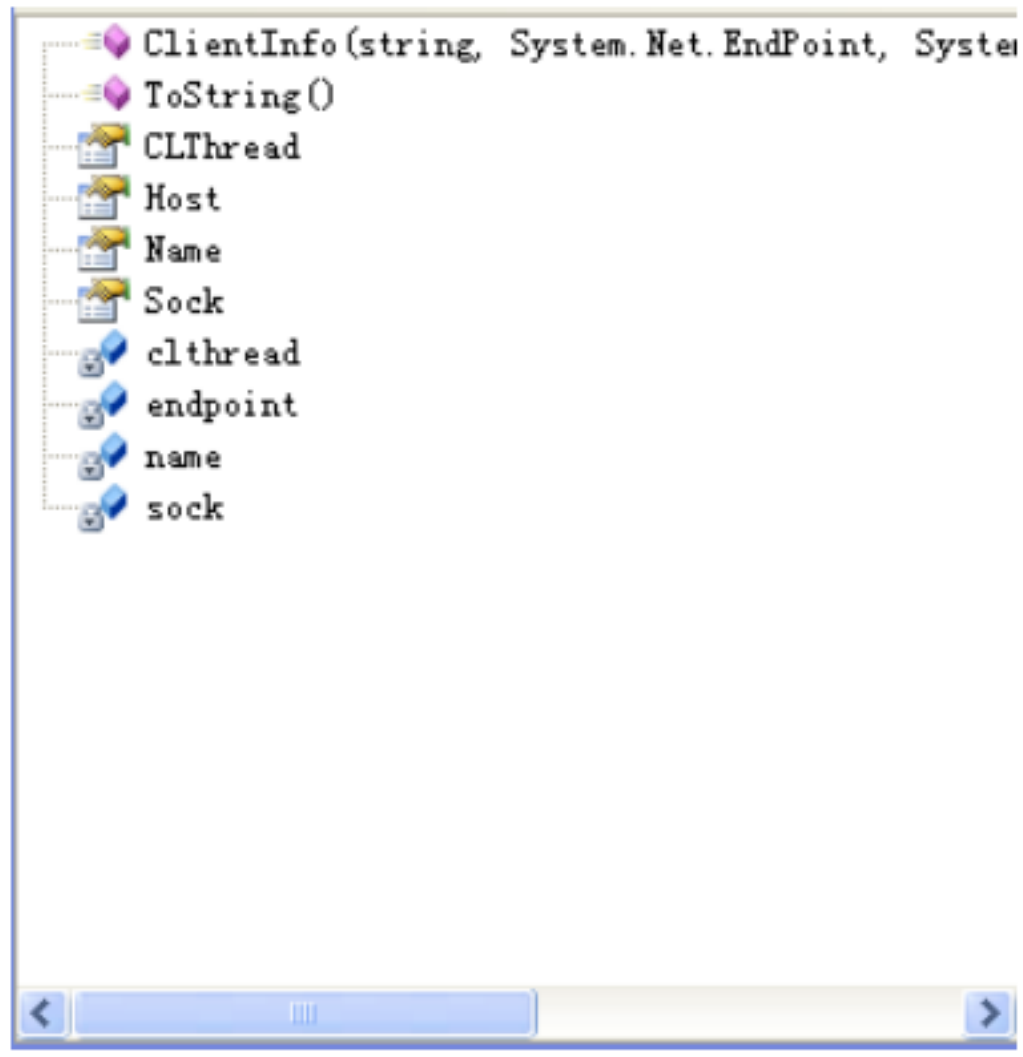
```
private void  StartListening()
        {
            byte[] ipadre =  new byte[] { 10, 82, 14, 47};
            IPAddress ip= new IPAddress(ipadre);
            m_Tcplisten =  new TcpListener(ip,m_Port);
            m_Tcplisten.Start();
            while  (true)
            {
                try
                {
                    Socket s = m_Tcplisten.AcceptSocket();
                    clientSocket = s;
                    m_serverThread = new Thread(new  ThreadStart(serviceClient)); //多线程 deal各个连接用
户的 socket
                    m_serverThread.Start();
                }
                catch (Exception  E)
                {
                    Console.WriteLine(E.ToString());
                }
            }
         }
```

如以上 startlistening 代码所示，监听创造一个连接客户端的套接字，再用多线程处理该连接，而服务器端则继续监听新的套接字。

这样主要的交易代码就可以放入 ServiceClient 这个函数中，当有新客户信息连入时，即可进行查询数据库，对比插入股票队列等工作

Class ClientInfo

//这个类记录了客户端的　　socket



数据访问层类图

Class ADOSQLserver

Private

dataSet

//ds 下可有 4 个 dataTable

userTable

stockTable

User_stockTable

tempTable

Public:

//// 验证用户信息

Bool CheckUserlogin(string usridstring password);

Bool CheckUserMoney(string userID);

Bool CheckUserStockCount(string userID);

//// 交易成功修改用户和股票信息

Void updateUserTable（Class stockData）

Void updateStockTable（Class stockData）

Void updateUser_stockTable(Class stockData)

/// 还未成功的交易放入临时表,

Void updateTemTable（Class stockData）

注意，每次交易成功要删除临时表的信息

Void deleteInfo（Class stockData）

Class stockData

订单号 public int ListID;

public int UsrID;

public string StockIndex;

public int Prince;

public int Count;

public bool Isbuy;

该类即为向服务端传送数据时的包

关于交易算法的详细设计

5.2 撮合算法

在前文中，我们已经提到了，撮合算法是整个交易所乃至整个证券仿真系统的核心部分。此算法的成功与否，直接影响着仿真系统是否能实现以及实现效率的高低。

按照真实的交易原则，撮合算法分为连续竞价和集中竞价两种方式。

下面我们将分别对这两种方式进行实现。

5.2.1 连续竞价

连续竞价是在绝大部分交易时间使用的撮合算法。

连续竞价原则：

1.) 价格优先原则：价格较高的买入申报优先于价格较低的买入申报，价格较低的卖出申报优先于价格较高的卖出申报。

2.) 时间优先原则：同价位申报、依照申报时序决定优先顺序，即买卖方向、价格相同的，先申报者先于后申报者。先后顺序按证券交易所主机接受申报的时间确定。

在正常情况下，买队列的第一笔（报价最高）的报价一定小于卖队列的第一笔（最低报价）的报价。此时不发生撮合。一旦买卖队列的价格发生了交叉，如图 2.3.1 所示，发生交叉的那部分就会进行撮合。

而事实上，由于每一笔新来的单子进入数列后都会触发一次比较，所以每次触发撮合都是由新单子促成的。称为“来一笔撮合一次”，也就是连续竞价。

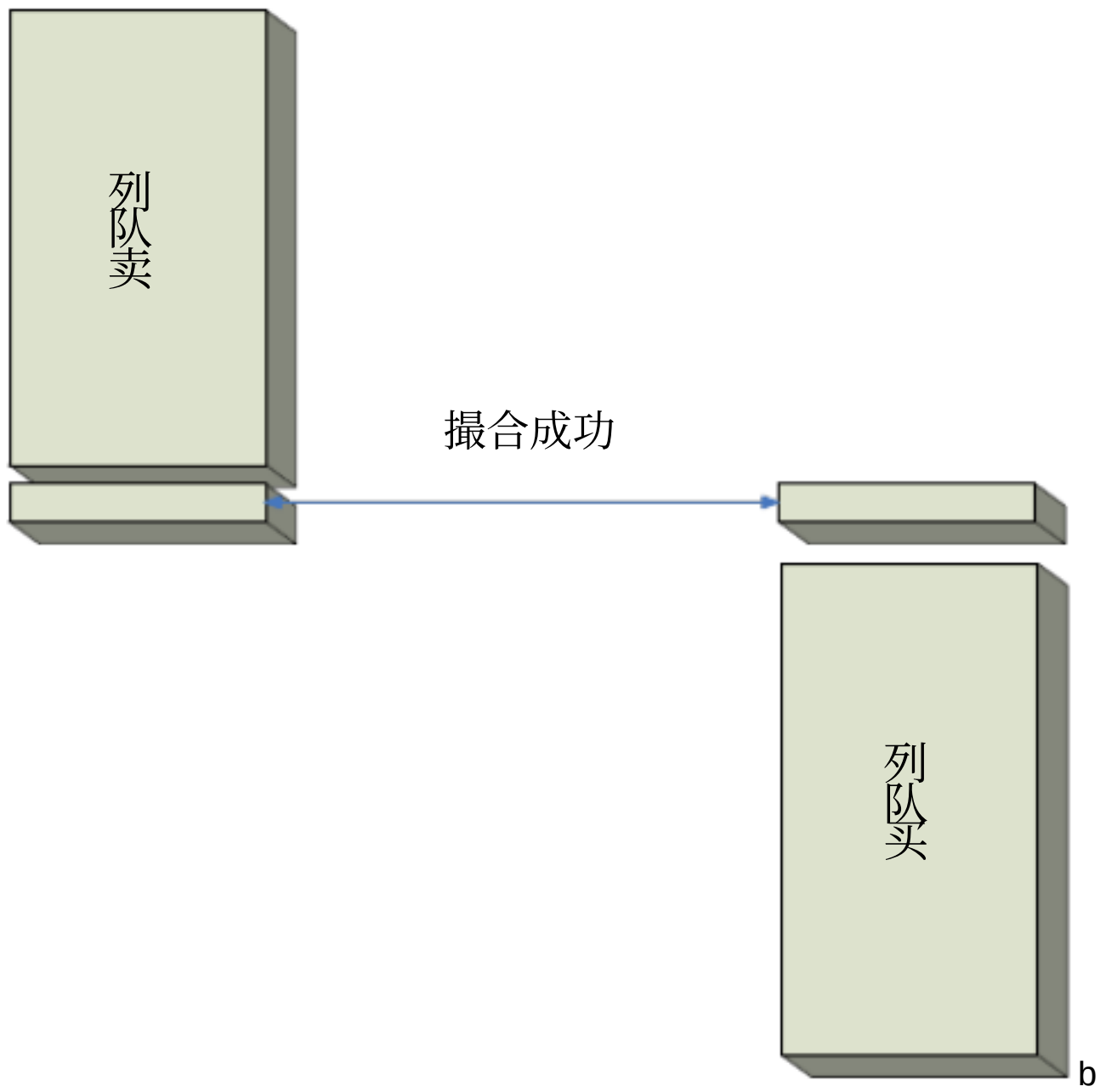


图 2.3.1

连续竞价算法描述：

首先设定 QueueStruct 结构为元素的买卖两个队列 BuyQueue和 SellQueue。

为了尽可能的提高效率，减少资源占用，我们用静态数组构建这两个队列。

其中 BuyQueue是时间优先、买价降序排序，而 SellQueue 是时间优先、卖价升序排序，在连续竞价条件下，可以保证 BuyQueue[0]的 price 小于 SellQueue[0] 的 price 。

连续竞价算法如下：

1.) 接收一个新单子 newlist ，判断 newlist 是买单还是卖单；
如果是买单，则转 2，如果是卖单，则转 B；

2.) 取卖单队列头 SellQueue[0] ，if SellQueue[0].price>newlist.price ，利用插入排序将 newlist 插入到买队列 BuyQueue中，转 1；

3.) if SellQueue[0].count>newlist.count ，newlist 完全撮合，SellQueue[0].count = SellQueue[0].count －

newlist.count ，转 2；

4.) if SellQueue[0].count<=newlist.count ，SellQueue[0] 撮合，并将 SellQueue[0] 从 SellQueue 队列中删除，newlist.count=newlist.count-SellQueue[0].count ，转 2；

5.) 取买单队列头 BuyQueue[0] ，if BuyQueue[0].price<newlist.price ，利用插入排序将 newlist 插入到卖队列 BuyQueue中，转 1；

6.) if BuyQueue[0].count>newlist.count ，newlist 完全撮合，BuyQueue[0].count ＝BuyQueue[0].count －newlist.count ，转 1；

7.) if BuyQueue[0].count<=newlist.count ，BuyQueue[0]撮合，并将 BuyQueue[0]从 BuyQueue队列中删除，newlist.count=newlist.count-BuyQueue[0].count ，转 5；

如下面流程 图 5.2.2 所示：

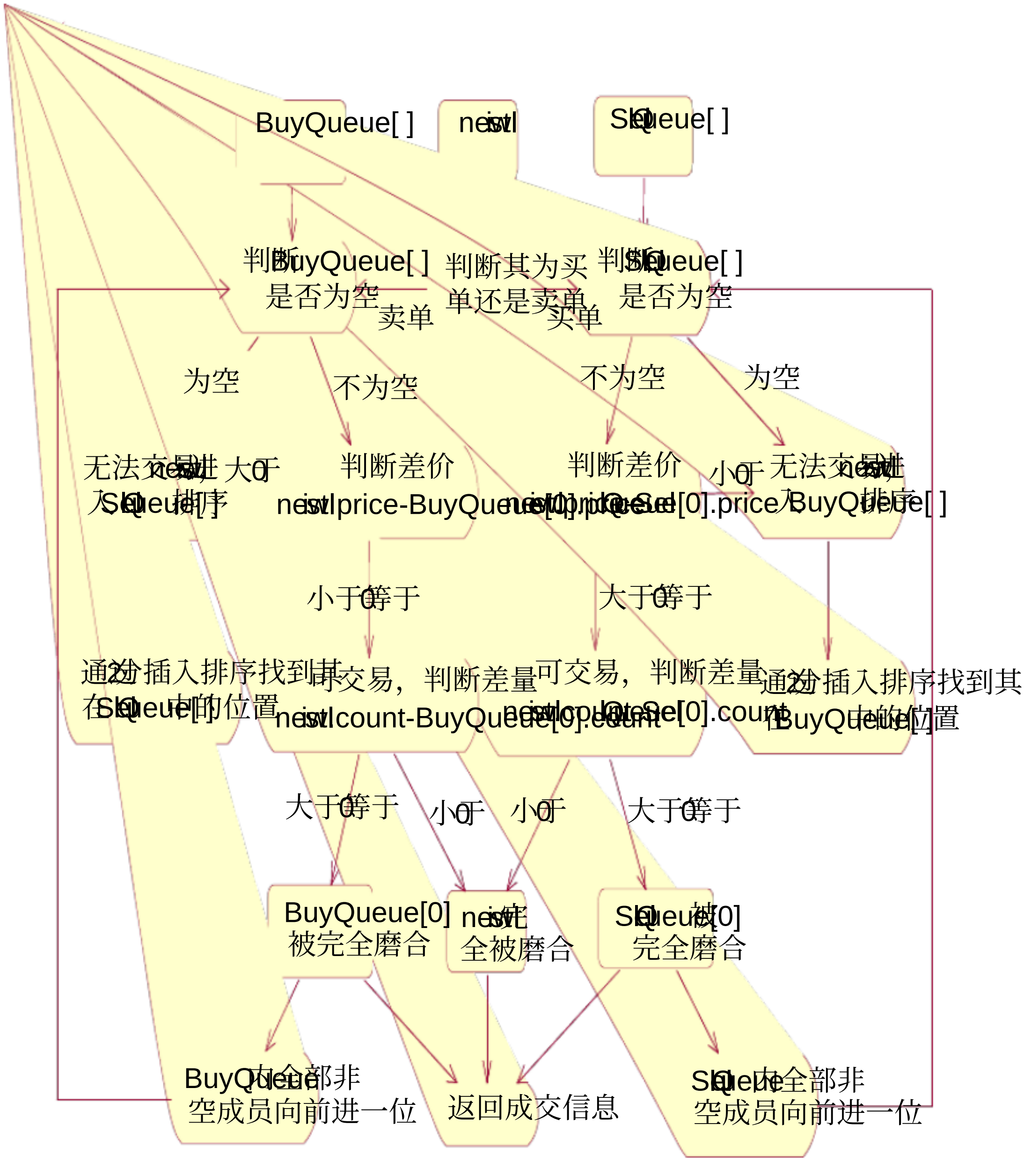

BuyQueue[ ]
SellQueue[ ]
判断BuyQueue[ ]是否为空
判断其为买单还是卖单
判断SellQueue[ ]是否为空
卖单
买单
为空
不为空
不为空
为空
判断差价
判断差价
小于0
大于0
等于0
小于0
大于0
等于0
通过插入排序找到其在SellQueue[ ]中的位置
通过插入排序找到其在BuyQueue[ ]中的位置
小于0
大于0
等于0
小于0
大于0
等于0
BuyQueue[0]被完全磨合
SellQueue[0]被完全磨合
BuyQueue[ ]内全部非空成员向前进一位
返回成交信息
SellQueue[ ]内全部非空成员向前进一位

图 3.2.2

5.2.2　集合竞价

集合竞价是指对所有有效委托进行集中处理，　深、沪两市的集合竞价时间为交易日上午 9：15 至 9：25。

集合竞价原则：

凡是高于开盘价的买单一定成交；

凡是低于开盘价的卖单一定成交；

凡是高于开盘价的卖单一定不成交；

凡是低于开盘价的买单一定不成交；

集合竞价分四步完成：

第一步：确定有效委托在有涨跌幅限制的情况下，有效委托是这样确定的：根据该只证券上一交易日收盘价以及确定的涨跌幅度来计算当日的最高限价、　最低限价。有效价格范围就是该只证券最高限价、最低限价之间的所有价位。

限价超出此范围的委托为无效委托，系统作自动撤单处理。

第二步：系统根据竞价规则自动确定集合竞价的成交价，　这个价格就是当日的开盘价，　所有高于开盘价的买盘和所有低开开盘价的卖盘均以此价格成交，集合竞价的成交价确定原则是：以此价格成交，能够得到最大成交量。

第三步：集中撮合处理所有的买委托按照委托限价由高到低的顺序排列，　限价相同者按照进入系统的时间先后排列；　所有卖委托按委托限价由低到高的顺序排列　，　限价相同者按照进入系统的时间先后排列。　依序逐笔将排在前面的买委托与卖委托配对成交，　即按照 " 价格优先，同等价格下时间优先 " 的成交顺序依次成交，直至成交条件不满足为止，　即不存在限价高于等于成交价的叫买委托、　或不存在限价低于等于成交价的叫卖委托。

所有成交都以同一成交价成交。　这同一成交价成交的买卖单一般量都是很大的，如图 3.2.3 所示

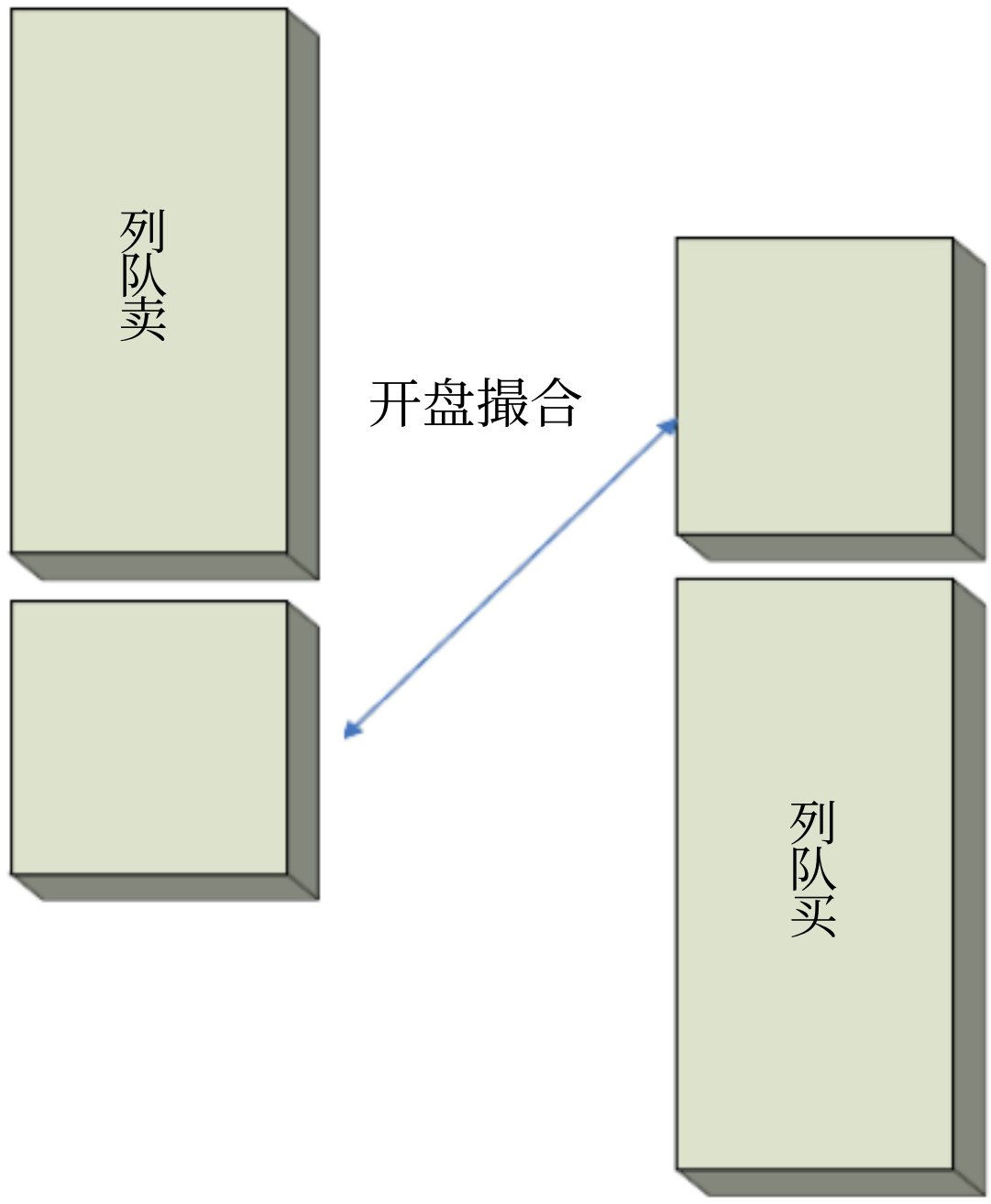


图 3.2.3 所示

第四步：行情揭示：

1.) 如该只证券的成交量为零，则将成交价位揭示为开盘价、最近成交价、最高价、最低价，并揭示出成交量、成交金额。

2.) 剩余有效委托中，实际的最高叫买价揭示为叫买揭示价，若最高叫买价不存在，则叫买揭示价揭示为空；实际的最低叫卖价揭示为叫卖揭示价，若最低叫卖价不存在，则叫卖揭示价揭示为空。

集合竞价中未能成交的委托，自动进入连续竞价。

按照这样的原则和要求，我们设计了如下的集合竞价撮合算法。如图 3.2.4 所示。

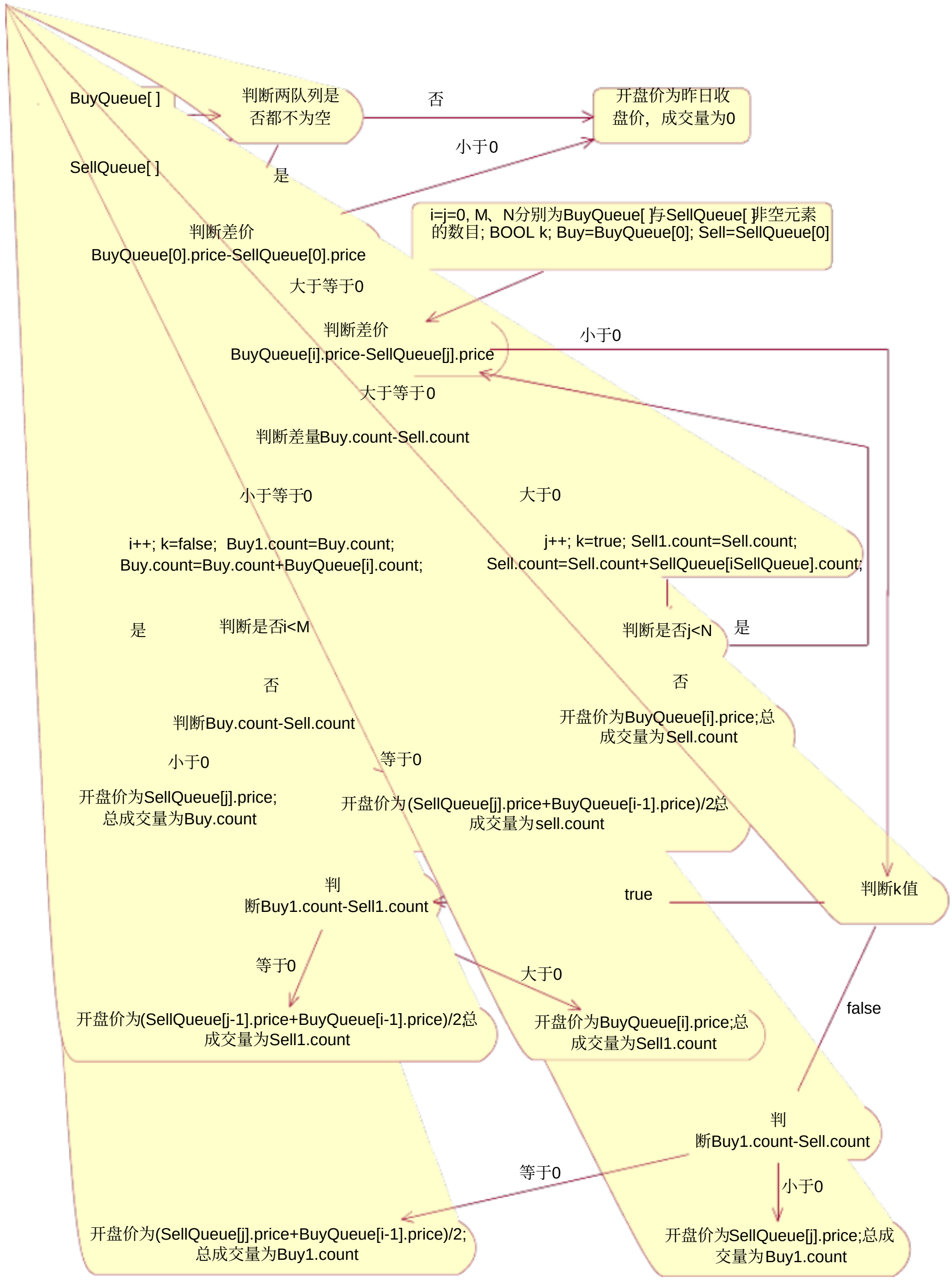

BuyQueue[ ]
SellQueue[ ]
判断两队列是否都不为空
否
开盘价为昨日收盘价，成交量为0
小于0
是
i=j=0, M、N分别为BuyQueue[ ]与SellQueue[ ]非空元素的数目; BOOL k; Buy=BuyQueue[0]; Sell=SellQueue[0]
判断差价
BuyQueue[0].price-SellQueue[0].price
大于等于0
判断差价
BuyQueue[i].price-SellQueue[j].price
小于0
大于等于0
判断差量Buy.count-Sell.count
小于等于0
大于0
i++; k=false; Buy1.count=Buy.count;
Buy.count=Buy.count+BuyQueue[i].count;
j++; k=true; Sell1.count=Sell.count;
Sell.count=Sell.count+SellQueue[iSellQueue].count;
是
判断是否i<M
判断是否j<N
是
否
否
判断Buy.count-Sell.count
开盘价为BuyQueue[i].price;总成交量为Sell.count
小于0
等于0
开盘价为SellQueue[j].price;总成交量为Buy.count
开盘价为(SellQueue[j].price+BuyQueue[i-1].price)/2总成交量为sell.count
判断Buy1.count-Sell1.count
true
判断k值
等于0
大于0
false
开盘价为(SellQueue[j-1].price+BuyQueue[i-1].price)/2总成交量为Sell1.count
开盘价为BuyQueue[i].price;总成交量为Sell1.count
判断Buy1.count-Sell.count
等于0
小于0
开盘价为(SellQueue[j].price+BuyQueue[i-1].price)/2;总成交量为Buy1.count
开盘价为SellQueue[j].price;总成交量为Buy1.count

图 3.2.4

集合竞价算法描述：

和连续竞价一样，首先设定 QueueStruct 结构为元素的买卖两个队列 BuyQueue 和 SellQueue 。

为了尽可能的提高效率，减少资源占用，我们用静态数组构建这两个队列。

其中 BuyQueue 是时间优先、买价降序排序，而 SellQueue 是时间优先、卖价升序排序。在开市到开盘这段时间内，买卖单已经分别进入了买卖队列内排好了序。

一旦宣布开盘，则触发集合撮合，如下：

判断两队列是否都不为空，如是，转 2；如否，转 21；

判断 BuyQueue[0].prince 与 SellQueue[0].prince 之差，如大于等于 0，转 3：如小于 0，转 21；

定义 int i=j=0 ；M、N 分别为买卖两队列非空元素的个数；BOOL k；QueueStruct Buy=BuyQueue[0] ；Sell=SellQueue[0] ；Buy1；Sell1 ；转 4；

判断 BuyQueue[i].prince 与 SellQueue[j].prince 之差，如大于等于 0，转 5：如小于 0，转 14；

判断 Buy.count 与 Sell.count 之差，如大于 0，转 6；如小于等于 0，转 9；

j++ ； k=true ； Sell1.count=Sell.count ；Sell.count=Sell.count+SellQueue[iSellQueue].count ；转 7；

判断 j 是否小于 N，如是，转 4；如不是，转 8；

开盘价为 BuyQueue[i].price ；总成交量为 Sell.count ；统计成交数据及回报，并返回；

i++ ； k=false ； Buy1.count=Buy.count ；Buy.count=Buy.count+BuyQueue[i].count ；转 10；

判断 i 是否小于 M，如是，转 4；如不是，转 11；

判断 Buy.count 与 Sell.count 之差，如小于 0，转 12；如等于 0，转 13；

开盘价为 SellQueue[j].price ；总成交量为 Buy.count ；统计成交数据及回报，并返回；

开盘价为 (SellQueue[j].price+BuyQueue[i -1].price)/2 ；总成交量为 sell.count ；统计成交数据及回报，并返回；

判断 k 值，如为 true ，转 15；如为 false ，转 18；

判断 Buy1.count 与 Sell1.count 之差，如大于 0，转 16；如小于 0，转 17；

开盘价为 BuyQueue[i].price ；总成交量为 Sell1.count ；统计成交数据及回报，并返回；

开盘价为 (SellQueue[j-1].price+BuyQueue[i-1].price)/2 ；总成交量为 Sell1.count ；统计成交数据及回报，并返回；

判断 Buy1.count 与 Sell.count 之差，如小于 0，转 19；如等于 0，转 20；

开盘价为 SellQueue[j].price ；总成交量为 Buy1.count ；统计成交数据及回报，并返回；

开盘价为 (SellQueue[j].price+BuyQueue[i -1].price)/2 ；总成交量为 Buy1.count ；统计成交数据及回报，并返回；

开盘价为昨日收盘价，成交量为 0；保留所有数据至开盘进入连续竞价撮合；

5.2.3 买卖队列排序

上面我们介绍了撮合算法的核心部分，但实际上在撮合前后都要对两个买卖队列进行一定的插入和排列处理，这在整个算法中也是很重要的部分。下面我们就来具体介绍一下。

对所有的排列和插入我们考虑了效率问题之后，最后统一使用了二分插入排序法。

在单子进入队列时，我们首先统计出当前队列中的非空数据个数，然后再通过新单子与当前队列中间值的价格比较，确定新单子在队列的前半部分还是后半部分，然后再取该区域中间值与之比较，直到确定新单子应在的位置。

如下列代码所示：

```
int low=0; int high=N-1;                    //N 为队列中非空元素的个数
 while(low<=high)
 {
       int m=(low+high)/2;
       if(newlist->price<SellQueue[m].price)
             high=m-1;
       else       low=m+1;
 }
 for(int i=N-1; i>=high+1; --i)
{
       SellQueue[i+1]=SellQueue[i];
}
 SellQueue[high+1]=*newlist;
```

这是卖队列的排序，对于买队列的排序与之相似，只是价格排列是由高到底。在这里不再赘述。

这种插入排序方法完全符合了撮合算法中价格优先、时间优先的要求，而且效率也是比较高的。

在集合竞价前和连续竞价后进行的插入排序都是这样进行的，而在集合竞价撮合之后，对两队列的重新排列，我们首先使用了 memset 函数将前面已全部成交的 t 个元素清空，然后将 t 到 N(原总非空元素个数)前移 t 位。

如下列代码所示：

```
for (int p=0; *t < *N; p++,*t++)
    {
        QueueStruct temp;
        temp=Queue[*t];
        Queue[*t]=Queue[p];
        Queue[p]=temp;
    }
```

5.2.4　撮合算法的运行机制

在交易所正常运行时，一天内分为开市、开盘、休市、复开、收市等　　5 个步骤。

开市：每天上午　9:15 开市。这时候，股民可以通过券商向交易所递单。同一只股票的买卖单开始分别进入这只股票的买卖队列中，　　但并不进行撮合。 该过程一直持续到　9:25。

开盘：每天上午　9:30 正式开盘。 9:25-9:30 为盘前处理，这段时间也就是集合竞价撮合算法运行的时间。　9:25，买卖两队列开市不再接收新的单子。新的单子这时都放在缓冲区中，直到两队列运行完整个集合竞价算法后开市重新进单时，再依序将单子从缓存区取出读入买卖队列进行连续竞价的撮合。　　买卖队列于 9:25 开始集合竞价，　运算完毕，　得出开盘价等所有统计数据并发布行情以及发送完所有回报之后，　重新接收单子进入该日正常交易中，　　使用连续竞价算法进行撮合。

休市：每天上午 11:00 休市。此时，所有的券商不再接受买卖单子，也不再给交易所递单。 交易所将此前已经收到的所有单子撮合之后处于休息阶段。 并进行各种当日前市盘点。

复开：每天下午 13:30 复开。此时，券商开市接收用户的递单，并将其送交交易所。交易所重新开市进入连续竞价撮合

收市：每天下午 15:00 收市。此时交易所不再接受任何单子， 准时进行最后一笔单子的撮合之后， 关闭撮合线程。 将最后得到的收盘价等数据统计发送完之后，当天的所有工作全部结束。

如图 3.2.5 所示。

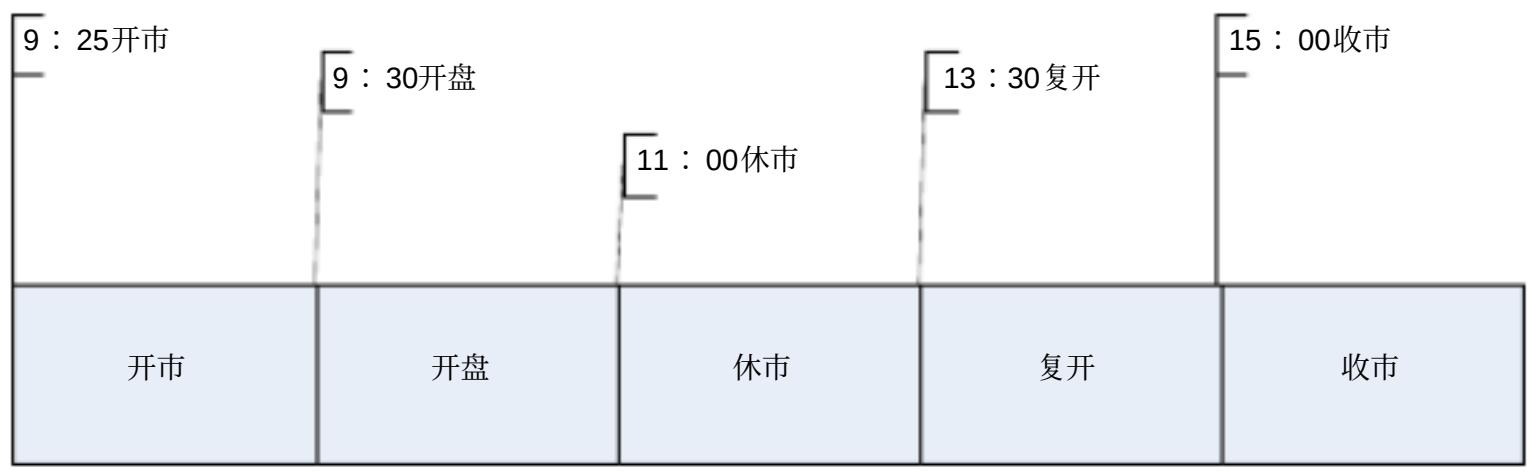


图 3.2.5

5.3 数据库设计

数据库服务器为　SQL SERVER 2005 EXPRESS

股票信息表

记录了股票的当前信息

表 - dbo.stockInfo

	列名	数据类型	允许空
🔑	stockIndex	nchar(10)	☐
	stockName	varchar(50)	☐
	stockCount	bigint	☐
▶	stockPrice	int	☐
			☐

用户信息表

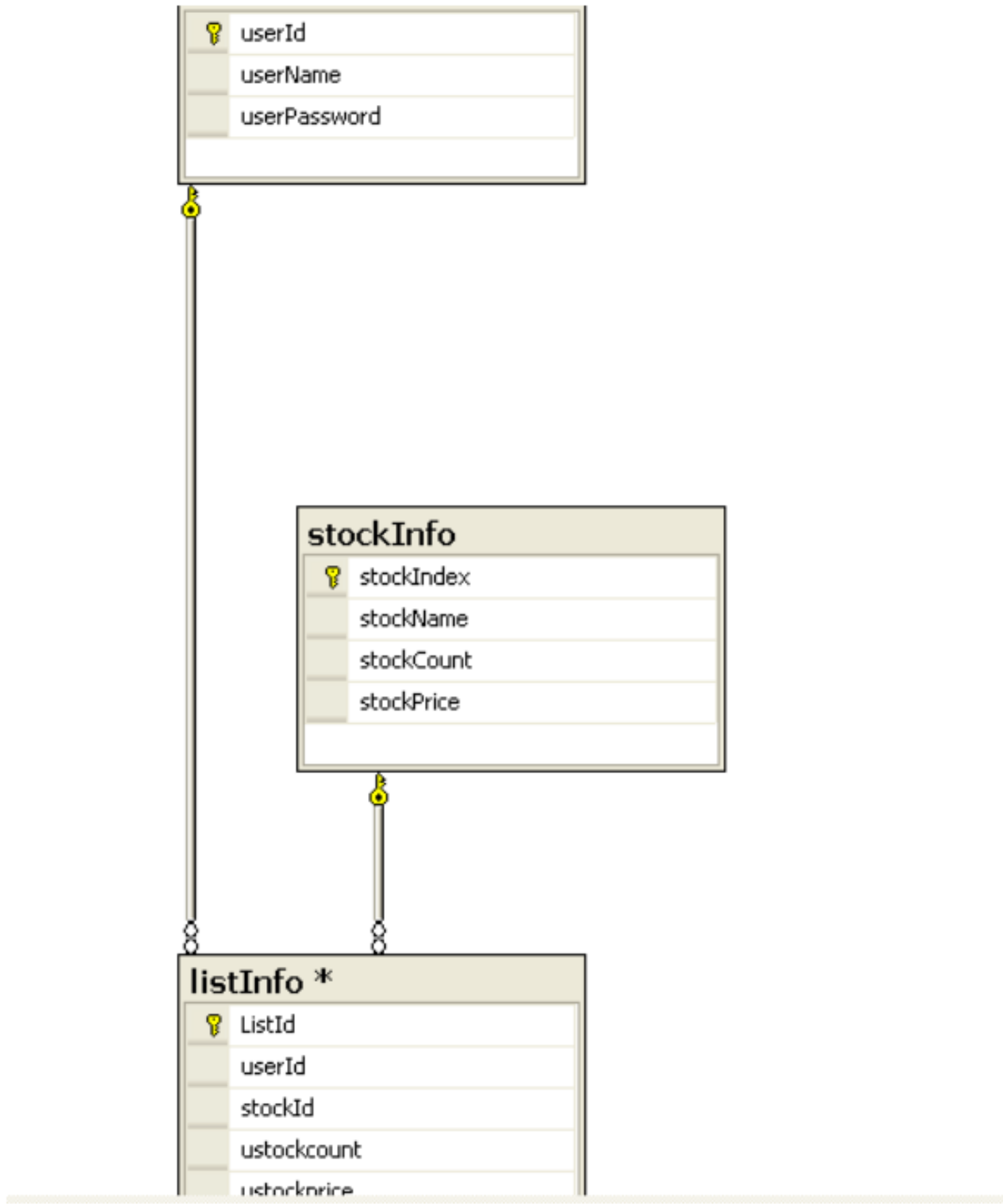
	列名	数据类型	允许空
▶🔑	userId	nchar(10)	☐
	userName	varchar(50)	☐
	userPassword	nchar(10)	☐
			☐

交易订单表

列名	数据类型	允许空
ListId	nchar(10)	☐
userId	nchar(10)	☐
stockId	nchar(10)	☐
ustockcount	int	☐
ustockprice	float	☐
		☐

其中 userid 和 stockid 是外键

关系图如下


userId
userName
userPassword
stockInfo
stockIndex
stockName
stockCount
stockPrice
listInfo *
ListId
userId
stockId
ustockcount

5.4 物理结构设计要点

给出本系统内所使用的每个数据结构中的每个数据项的存储要求，访问方法、存取单位、存取的物理关系（索引、设备、存储区域）、设计考虑和保密条件。

5.5 数据结构与程序的关系

说明各个数据结构与访问这些数据结构的各个程序之间的对应关系，可采用如下的矩阵图的形式：

	程序 1	程序 2	. . . . .	程序 m
数据结构 1	√			
数据结构 2		√		
⋮				
数据结构 n		√		√

6 系统出错处理设计

6.1 出错信息

用一览表的方式说明每种可能的出错或故障情况出现时，系统输出信息的形式、含意及处理方法。

6.2 补救措施

说明故障出现后可能采取的变通措施，包括：

a. 后备技术： 说明准备采用的后备技术， 当原始系统数据万一丢失时启用的副本的建立和启动的技术，例如周期性把磁盘信息记录到磁带上去就是对于磁盘媒体的一种后备技术；

b. 降效技术： 说明准备采用的后备技术， 使用另一个效率稍低的系统或方法来求得所需结果的某些部分，例如一个自动系统的降效技术可以是手工操作和数据的人工记录；

c. 恢复及再启动技术： 说明将使用的恢复再启动技术， 使软件从故障点恢复执行或使软件从头开始重新运行的方法。

6.3 系统维护设计

说明为了系统维护的方便而在程序内部设计中作出的安排， 包括在程序中专门安排用于系统的检查与维护的检测点和专用模块。

(未全部完成，在项目进行过程中完善）