**目录**

[一、 需求分析 1](#_Toc186373791)

[(一) 题目要求 1](#_Toc186373792)

[(二) 需求分析 1](#_Toc186373793)

[(三) 交易流程 2](#_Toc186373794)

[(四) 子模块功能 4](#_Toc186373795)

[二、 数据库设计 6](#_Toc186373796)

[(一) 关系模式设计 6](#_Toc186373797)

[(二) 表设计 6](#_Toc186373798)

[(三) 触发器设计 10](#_Toc186373799)

[(四) 视图设计 11](#_Toc186373800)

[(五) 数据转储 12](#_Toc186373801)

[三、 程序设计 13](#_Toc186373802)

[(一) 技术选型 13](#_Toc186373803)

[(二) 实体类设计 13](#_Toc186373804)

[(三) 服务设计 15](#_Toc186373805)

[四、 界面设计 21](#_Toc186373806)

[(一) 组件设计 21](#_Toc186373807)

[(二) 登录/注册页面 22](#_Toc186373808)

[(三) 个人中心页面 22](#_Toc186373809)

[(四) 表单页面 23](#_Toc186373810)

[(五) 选择组件 25](#_Toc186373811)

[(六) 错误页面 27](#_Toc186373812)

[五、 系统测试 28](#_Toc186373813)

[(一) 登录注册测试 28](#_Toc186373814)

[(二) 个人信息测试 28](#_Toc186373815)

[(三) 模拟交易测试 29](#_Toc186373816)

[(四) 删改测试 32](#_Toc186373817)

[六、 总结与心得 34](#_Toc186373818)

[七、 参考资料 35](#_Toc186373819)

# 需求分析

## 题目要求

建立一个零件交易中心管理系统, 编写应用程序完成系统开发。

1. **建立基本表：**
2. 零件信息（零件编号，颜色，零件名，重量，简介）；
3. 供应商信息（供应商编号，供应商名，地址，联系电话，简介）；
4. 顾客信息（顾客编号，顾客姓名，地址，联系电话）；
5. 供应信息（供应商和零件之间的联系）（零件编号，供应商编号，价格，数量）；
6. 求购信息（顾客和零件之间的联系）（顾客编号，零件编号，价格，数量）；
7. 交易（顾客编号，供应商编号，零件编号，价格，数量）；
8. 协议书（顾客编号，供应商编号，零件编号，价格，数量，顾客签名，供应商签名）。
9. **系统应实现下述功能：**
10. 供应商：要求实现供应商注册、注销、修改个人信息、增加、删除、修改供应的零件；
11. 顾客：要求实现顾客注册、注销、修改个人信息、增加、删除、修改需求信息；
12. 交易员：依据双方的信息，提出交易建议，供应商和顾客达成协议后完成交易；
13. 系统维护。

## 需求分析

作为一个零件交易管理子系统，应给用户提供方便、友好而简洁的界面应用，应提供顾客和供应商之间完成零件交易的功能，并能记录顾客、供应商、零件的信息，以便于交易员查看、分析及管理。

分析题目，得出系统具有下述5项需求：

1. **用户管理**：实现顾客、供应商、交易员、管理员的注册、登录、信息管理；
2. **增改删查**：零件信息、顾客信息、顾客需求、供应商信息、供应信息、交易员信息、交易信息；
3. **筛选与查询**：对于零件信息、顾客需求、供应信息的模糊查询，并可根据个别字段来筛选信息，便于查询；
4. **供需管理**：顾客和供应商可以发布需求和供应信息，平台管理供需信息，整合数据送交易员处理，方便其分析供需情况；
5. **交易管理**：顾客、供应商和交易员通过交易平台发起交易建议，完成交易协议，完成零件交易。

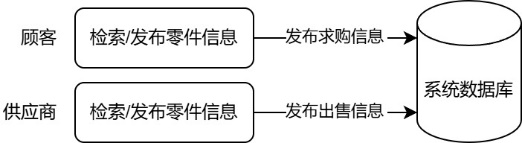
针对以上需求，开发出满足上述需求的零件交易中心系统，设计网页界面和后台逻辑，在简化用户操作的同时满足用户对于系统的各种操作需求。

## 交易流程

分析需求，本系统的核心功能是整合顾客和供应商的供需信息，让交易员分析并提出交易协议，以本系统为媒介让双方促成交易。零件交易的具体流程如下：

1. **顾客和供应商发布零件和供需信息**

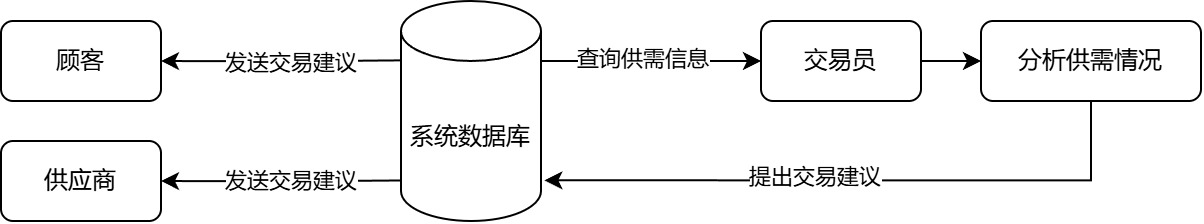
顾客和供应商根据各自的需求和供应情况，在系统检索或发布所需的零件信息、求购信息和供应信息。



**图1-1　顾客和供应商发布信息**

1. **交易员分析供需关系提出交易建议**

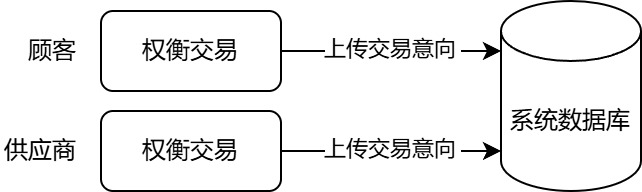
交易员通过分析系统的需求表和供应表，查找符合供需关系的记录，并向提出交易建议，并报送顾客和供应商。



**图1-2　交易员提出交易建议**

1. **顾客和供应商响应交易建议：**

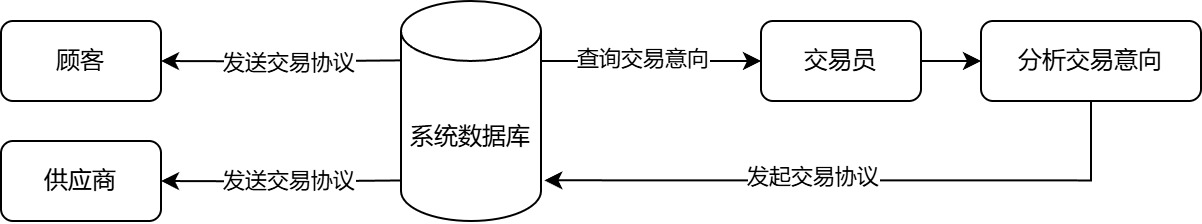
顾客和供应商根据交易员提出的建议，各自权衡做出决定，向平台提交交易意向并报送交易员。



**图1-3　顾客和供应商响应交易建议**

1. **达成交易意向，交易员发起交易协议：**

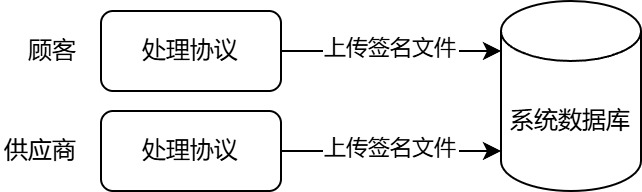
交易员从系统得到顾客和供应商对提出建议的交易意向，若都为同意，则发起交易协议，报送顾客和供应商。



**图1-4　交易员发起交易协议**

1. **顾客和供应商提交签名达成交易协议：**

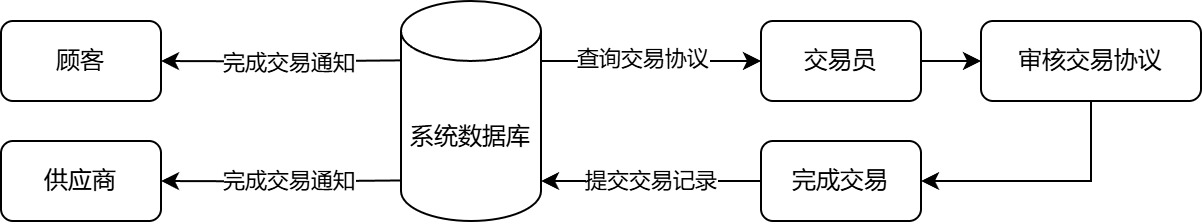
顾客和供应商向系统提交签名文件，供交易员审核。



**图1-5　顾客和供应商提交签名**

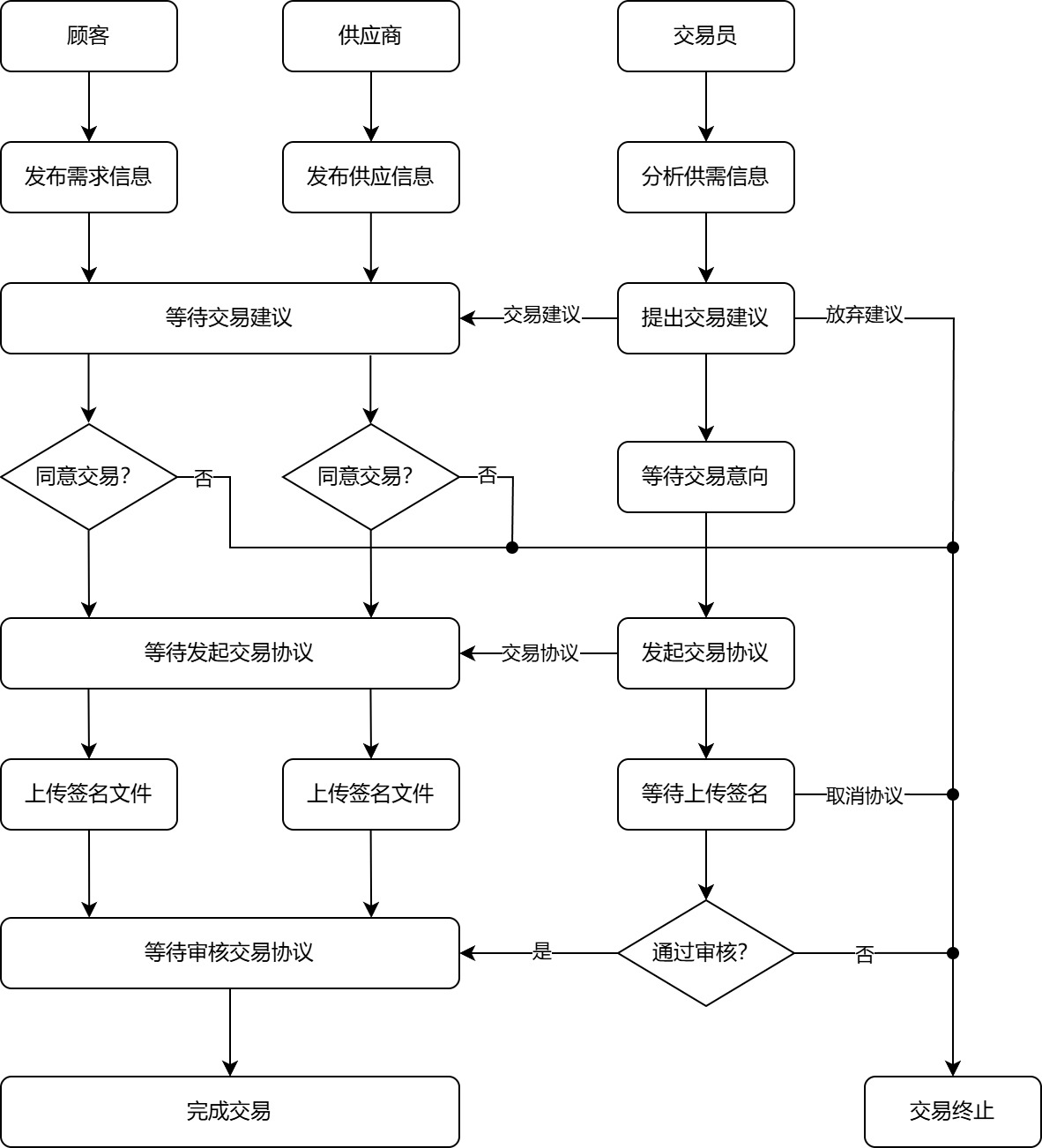
1. **交易员审核协议，完成交易：**

交易员审核顾客和供应商的签名文件，若通过审核，则向系统提交完成交易，并通知顾客和供应商。



**图1-6　交易员审核协议并完成交易**

　　综上，零件交易的总流程图如下：



**图1-7　零件交易总流程图**

## 子模块功能

为确保系统设计的低耦合和高可维护性，对系统划分为若干子模块，同时根据用户角色设置不同种权限，确保数据的安全性。

1. **登录/注册模块：**

该模块负责既有用户的登录和新用户的注册功能。登录和注册时需要向数据库查询用户名和密码是否存在和正确。其中，以安全性起见，不允许新用户以管理员身份注册新账号。

1. **个人信息模块：**

该模块负责用户更改昵称、修改密码、账号登出和注销功能，其中在修改关键字段时应确保唯一性和安全性。

1. **表单交互模块：**

该模块负责表格的显示、筛选、查询、分页和导出功能。并注意需要根据用户的权限灵活控制表单各字段的显示和查询。同时，限定各类别用户的权限如下：

1. **顾客：**
2. 查询/发布零件信息：名称、颜色、重量和简介；
3. 查询/发布/修改/删除需求信息：零件编号、预算、数量；
4. 处理交易建议：同意/拒绝交易员发起的交易建议；
5. 上传签名文件：上传签名文件；
6. 查询历史交易记录。
7. **供应商：**
8. 查询/发布零件信息：名称、颜色、重量和简介；
9. 查询/发布/修改/删除供应信息：零件编号、报价、数量；
10. 处理交易建议：同意/拒绝交易员发起的交易建议；
11. 上传签名文件：上传签名文件；
12. 查询历史交易记录。
13. **交易员：**
14. 查询供需信息：分析供求分析；
15. 发起/放弃交易建议：名称、顾客、供应商、报价；
16. 发起/审批/取消/交易协议：名称、审批意见；
17. 查询历史交易记录。
18. **管理员：**

新增/删除/查询/修改所有表数据。

# 数据库设计

## 关系模式设计

根据题目要求和第一章的分析，利用数据库的设计方法，可以设计详细的关系模式。由于本应用中存在一个实体有多个外键的情况，若级联删除其中一个外键对应的记录，可能会影响其他外键对应的记录。因此本程序中，记录的删除使用软删除：设立一个名为删除时间的属性，若删除时间不为空，值为删除的时间戳；否则，记录未被删除。这样做的好处是在顾客、供应商或交易员注销账号时，与其关联的已完成交易并不会被级联删除，也能保存其他用户的记录。

以下是关系模式的详细设计：

1. **顾客**（顾客ID，姓名，密码，电话，地址，注册时间，删除时间）；
2. **供应商**（供应商ID，姓名，密码，电话，地址，注册时间，删除时间）；
3. **交易员**（交易员ID，姓名，密码，电话，地址，注册时间，删除时间）；
4. **管理员**（管理员ID，姓名，密码，电话，注册时间，删除时间）；
5. **零件**（零件ID，名称，颜色，重量，简介，删除时间）；
6. **需求**（需求ID，名称，预算，数量，零件ID，顾客ID，删除时间）；
7. **供应**（供应ID，名称，报价，数量，零件ID，供应商ID，删除时间）；
8. **交易建议**（交易建议ID，名称，报价，数量，顾客意向，供应商意向，交易员意向，需求ID，供应ID，交易员ID，删除时间）；
9. **协议**（协议ID，名称，顾客签名，供应商签名，审批意见，交易建议ID，删除时间）；
10. **交易记录**（交易记录ID，名称，完成时间，协议ID，删除时间）。

显然，以上关系模式的规范程度都达到了BCNF级别。

## 表设计

本交易系统采用MySQL作为数据库管理系统，因为MySQL具有高效的性能、良好的稳定性以及广泛的社区支持，能够满足高并发和大规模数据存储的需求。其支持事务、锁机制和复杂查询，使得数据一致性和安全性得以保障。以下是10张基本表的具体设计。

1. **Customers**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | Password | VARCHAR(16) | NOT NULL |
| 4 | Phone | VARCHAR(15) | - |
| 5 | Address | VARCHAR(128) | - |
| 6 | RegisterTime | DATETIME | NOT NULL |
| 7 | DeleteTime | DATETIME | - |

**表2-1　Customers表设计**

1. **Suppliers**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | Password | VARCHAR(16) | NOT NULL |
| 4 | Phone | VARCHAR(15) | - |
| 5 | Address | VARCHAR(128) | - |
| 6 | Brief | VARCHAR(255) | - |
| 7 | RegisterTime | DATETIME | NOT NULL |
| 8 | DeleteTime | DATETIME | - |

**表2-2　Suppliers表设计**

1. **Traders**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | Password | VARCHAR(16) | NOT NULL |
| 4 | Phone | VARCHAR(15) | - |
| 5 | Brief | VARCHAR(255) | - |
| 6 | RegisterTime | DATETIME | NOT NULL |
| 7 | DeleteTime | DATETIME | - |

**表2-3　Traders表设计**

1. **Administrators**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | Password | VARCHAR(16) | NOT NULL |
| 4 | Phone | VARCHAR(15) | - |
| 5 | RegisterTime | DATETIME | NOT NULL |
| 6 | DeleteTime | DATETIME | - |

**表2-4　Administrators表设计**

1. **Components**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | Color | VARCHAR(16) | NOT NULL |
| 4 | Weight | FLOAT | NOT NULL |
| 5 | Brief | VARCHAR(255) | - |
| 6 | DeleteTime | DATETIME | - |

**表2-5　Components表设计**

1. **Needs**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | Price | DECIMAL(10, 2) | NOT NULL, > 0 |
| 4 | Amount | INT | NOT NULL, > 0 |
| 5 | ComponentId | INT | FOREIGN KEY |
| 6 | CustomerId | INT | FOREIGN KEY |
| 7 | DeleteTime | DATETIME | - |

**表2-6　Components表设计**

1. **Supplies**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | Price | DECIMAL(10, 2) | NOT NULL, > 0 |
| 4 | Amount | INT | NOT NULL, > 0 |
| 5 | ComponentId | INT | FOREIGN KEY |
| 6 | SupplierId | INT | FOREIGN KEY |
| 7 | DeleteTime | DATETIME | - |

**表2-7　Supplies表设计**

1. **TradeSuggests**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | Price | DECIMAL(10, 2) | NOT NULL, > 0 |
| 4 | Amount | INT | NOT NULL, > 0 |
| 5 | CustomerIntention | BOOLEAN | - |
| 6 | SupplierIntention | BOOLEAN | - |
| 7 | TraderIntention | BOOLEAN | - |
| 8 | TraderId | INT | FOREIGN KEY |
| 9 | NeedId | INT | FOREIGN KEY |
| 10 | SupplyId | INT | FOREIGN KEY |
| 11 | DeleteTime | DATETIME | - |

**表2-8　TradeSuggests表设计**

1. **Agreements**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | CustomerSignature | BLOB | - |
| 4 | SupplierSignature | BLOB | - |
| 5 | TraderIntention | BOOLEAN | - |
| 6 | SuggestId | INT | FOREIGN KEY |
| 7 | DeleteTime | DATETIME | - |

**表2-9　Agreements表设计**

1. **Trades**

|  |  |  |  |
| --- | --- | --- | --- |
| **序号** | **字段名** | **类型** | **完整性约束** |
| 1 | Id | INT | PRIMARY KEY |
| 2 | Name | VARCHAR(64) | NOT NULL |
| 3 | CompleteTime | DATETIME | NOT NULL |
| 4 | AgreementId | INT | FOREIGN KEY |
| 5 | DeleteTime | DATETIME | - |

**表2-10　Trades表设计**

## 触发器设计

虽然使用软删除能规避级联删除使得关联记录丢失的影响，但是在部分情况下，系统依旧期望能实现部分的级联删除。比如当顾客注销账号时，其名下的所有需求信息也应该被一同删除。为实现该功能，可以在表上设置触发器，当DeleteTime字段被设置时，自动更新指定表的指定记录的DeleteTime被同时赋值。

以下是设置在Customers表和Suppliers表上触发器的定义：

1. **AFTER\_DELETE\_CUSTOMER**

DELIMITER $$

CREATE TRIGGER AFTER\_DELETE\_CUSTOMER

AFTER UPDATE ON Customers

FOR EACH ROW

BEGIN

IF OLD.DeleteTime IS NULL OR OLD.DeleteTime <> NEW.DeleteTime THEN

UPDATE Needs

SET DeleteTime = NEW.DeleteTime

WHERE CustomerId = NEW.Id;

END IF;

END$$

DELIMITER ;

1. **AFTER\_DELETE\_SUPPLIER**

DELIMITER $$

CREATE TRIGGER AFTER\_DELETE\_SUPPLIER

AFTER UPDATE ON Suppliers

FOR EACH ROW

BEGIN

    IF OLD.DeleteTime IS NULL OR OLD.DeleteTime <> NEW.DeleteTime THEN

        UPDATE Supplies

        SET DeleteTime = NEW.DeleteTime

        WHERE SupplierId = NEW.Id;

    END IF;

END$$

DELIMITER ;

需要注意的是，表达式OLD.DeleteTime <> NEW.DeleteTime当且仅当运算符两侧类型都为NULL或DATETIME时才返回TRUE或FALSE，因此需要先判定OLD.DeleteTime的值是否为NULL，以保证触发器按预期工作。

## 视图设计

基本表组成了数据库的模式，但是对于应用程序而言，直接从基本表获取的数据并非易于使用。如管理员需要查看所有用户的信息，但是各类用户的记录分散在不同表中；交易员需要直观地分析平台中同一零件的供需关系，而供需关系含在各自的表中；应用程序需要展示交易的价格和数量，需要根据外键反复地从数据库查询记录。为减轻应用程序的设计负担和增强数据库系统的逻辑独立性，有必要根据部分需求建立视图。

为实现能在一张表内展示所有用户的信息，需要定义视图V\_UserInfos来对多张用户表的记录的部分字段取并集；为直观地展示平台中同一零件的供需关系，需要定义视图V\_TradableInfos对两张表进行连接查询；对于含有外键的实体，都各自定义视图来获取被参照实体的目标数据。以下是定义的各视图的关系模式：

1. **用户信息**（ID，用户ID，姓名，电话，注册时间，删除时间，用户类型）；
2. **供需信息**（ID，零件ID，零件名称，需求ID，需求名称，需求预算，需求数量，供应ID，供应名称，供应报价，供应数量）；
3. **需求**（需求ID，名称，预算，数量，零件ID，零件名称，顾客ID，顾客名）；
4. **供应**（供应ID，名称，报价，数量，零件ID，零件名称，供应商ID，供应商名）；
5. **交易建议**（交易建议ID，名称，报价，数量，顾客意向，供应商意向，交易员意向，需求ID，需求名称，供应ID，供应名称，交易员ID，交易员姓名，顾客ID，供应商ID，零件名称）；
6. **协议**（协议ID，名称，顾客签名，供应商签名，审批情况，价格，数量，交易建议ID，零件名称，顾客ID，需求名称，供应商ID，供应名称，交易员ID，交易员姓名）；
7. **交易**（交易ID，名称，完成时间，成交价，数量，协议ID，零件名称，需求名称，顾客ID，供应名称，供应商ID，交易员ID，交易员姓名）。

其中用户信息和供需信息是不可更新视图，其余的皆为可更新视图。但是为了防止出现预期外的行为，对其的更新操作应当根据主键修改对应的底层表。

## 数据转储

为保证数据的安全性，系统引入数据转储来实现备份功能。本系统使用mysqldump工具实现数据库的转储，命令格式如下：

mysqldump [选项] -u 用户名 -p[密码] 数据库名 [表名...] > 备份文件.sql

应用程序可以设立后台线程，定期执行命令，实现定期转储。有关这部分的具体设计详见第三章。

# 程序设计

## 技术选型

* 前端框架：ASP.NET Core Blazor
* 后端框架： .NET Core 8.0
* DBMS： MySQL 8.0.40
* 引用库： Microsoft.EntityFrameworkCore、BootstrapBlazor
* 开发平台： Microsoft Visual Studio 2022

**说明：**根据个人对技术的熟悉程度选择了上述技术。Blazor作为新兴的网页开发框架，能轻松地开发出高性能、美观而实用的网页；.NET Core能提供高效、稳定的服务端逻辑支持；EF Core可以提供数据对象模型的高效转换和SQL语句生成功能；BootstrapBlazor提供了Blazor丰富且高效的组件库，便于快速开发美观的页面。

## 实体类设计

本系统使用EF Core的代码优先模式管理ORM模型，需要编写与表字段一致的实体类。由于篇幅有限，此处仅展示部分重要类型的实现。

利用面向对象的设计思想，提取基本表的共性，剖析类的继承关系，设计抽象基类EntityBase，并要求有关数据库的所有实体类型都继承自之：

Public abstract class EntityBase

{

[ReadOnly(true)]

[Display(Name = "ID")]

public int Id { get; set; }

}

基于基本表的实体类都继承自ModelBase类，包含属性Name和DeleteTime。枚举ModelType指示当前实体所属的类别。

public abstract class ModelBase : EntityBase

{

[Display(Name = "名称")]

[Required(ErrorMessage = "{0}不得为空")]

[StringLength(64, ErrorMessage = "长度不得超过64")]

public string Name { get; set; }

public DateTime? DeleteTime { get; set; }

public abstract ModelType ModelType { get; }

public ModelBase()

{

Name = "";

}

}

关于用户的实体类都继承自UserBase类，包含属性Password、Phone和RegisterTime。同时，重写基类Object的Equals和GetHashCode方法以实现正确的用户比较。

public abstract class UserBase : ModelBase

{

[Required(ErrorMessage = "{0}不得为空")]

[StringLength(16, ErrorMessage = "长度不得超过16")]

public string Password { get; set; }

[StringLength(15, ErrorMessage = "长度不得超过15")]

public string? Phone { get; set; }

public DateTime RegisterTime { get; set; }

public UserBase()

{

Password = "";

}

public override bool Equals(object? obj)

{

if (obj is UserBase other)

{

return other.Id == Id && ModelType == other.ModelType;

}

return false;

}

public override int GetHashCode()

=> HashCode.Combine(Id, ModelType);

}

基于视图的实体类都继承自ViewModelBase<TModel>类，泛型参数TModel指示该类实际映射的基本表实体类型。接口ICustomIdProvider约束了类必须提供类的主键格式化函数，诸如主键为5的Customer实例的CustomId值为"CUS0005"。

public abstract class ViewModelBase<TModel> : EntityBase, ICustomIdProvider where TModel : ModelBase, ICustomIdProvider

{

[Display(Name = "名称")]

[Required(ErrorMessage = "{0}不得为空")]

[StringLength(64, ErrorMessage = "长度不得超过64")]

public string Name { get; set; }

public virtual string CustomId

=> TModel.GetCustomId(Id);

public static string GetCustomId(int? id)

=> TModel.GetCustomId(id);

public ViewModelBase()

{

Name = "";

}

}

其余的实体类的实现都比较简单，在此不多赘述，具体实现请参见随附代码文件。

## 服务设计

在使用Blazor构建的网页应用程序中，如果组件需要访问应用程序级别共享的数据，通常需要向程序注册服务。本系统设计了三个服务：数据库连接服务、在线用户管理服务和数据转储服务。

1. **数据库连接**

在程序启动时注册并配置数据库连接服务，需要设计类CenterDbContext并继承EF Core的DbContext类来实现数据库连接的管理和属性的映射。以下是CenterDbContext类的关键部分实现：

public class CenterDbContext : DbContext

{

public DbSet<Customer> Customers { get; protected set; }

public DbSet<Supplier> Suppliers { get; protected set; }

public DbSet<Trader> Traders { get; protected set; }

public DbSet<Administrator> Administrators { get; protected set; }

public DbSet<Component> Components { get; protected set; }

public DbSet<Supply> Supplies { get; protected set; }

public DbSet<Need> Needs { get; protected set; }

public DbSet<Agreement> Agreements { get; protected set; }

public DbSet<Trade> Trades { get; protected set; }

public DbSet<TradeSuggest> TradeSuggests { get; protected set; }

public DbSet<NeedVM> NeedVMs { get; protected set; }

public DbSet<SupplyVM> SupplyVMs { get; protected set; }

public DbSet<TradableInfo> TradableInfos { get; protected set; }

public DbSet<TradeSuggestVM> TradeSuggestVMs { get; protected set; }

public DbSet<AgreementVM> AgreementVMs { get; protected set; }

public DbSet<TradeVM> TradeVMs { get; protected set; }

public DbSet<UserInfo> UserInfos { get; protected set; }

// 后续代码略

}

　　需要映射的基本表和视图在本类中定义，实体的映射规则在程序集中另作配置，EF Core加载会自动查找实体类对应的映射规则。在应用程序启动时，通过appsettings.json文件获取默认连接字符串，并注册数据库连接服务：

builder.Services.AddDbContext<CenterDbContext>(options =>

options.UseMySql(ServerVersion.AutoDetect(builder.Configuration.GetConnectionString("DefaultConnection"))));

在组件需要使用该服务时注入之：

[Inject]

[NotNull]

CenterDbContext? DbContext { get; set; }

1. **在线用户管理服务**

本应用程序的部分组件需要修改用户的在线状态，因此需要设计一个在线用户管理服务供有需要的组件使用。本服务采用了非常简单的实现来管理用户的在线状态，以保证同一用户不会多次登录，实现用户访问的安全性。这里使用了线程安全的哈希集合，保证并发条件下的正确操作。

public class OnlineUsersService

{

protected ConcurrentHashSet<UserBase> OnlineUsersSet { get; } = new ConcurrentHashSet<UserBase>();

public bool Login(UserBase user)

{

return OnlineUsersSet.Add(user);

}

public bool IsOnline(UserBase user)

{

return OnlineUsersSet.Contains(user);

}

public bool Logout(UserBase user)

{

return OnlineUsersSet.TryRemove(user);

}

}

在应用程序启动时注册服务：

builder.Services.AddSingleton<OnlineUsersService>();

在组件需要使用服务时，注入该服务：

[Inject]

[NotNull]

OnlineUsersService? OnlineUsersService { get; set; }

1. **数据库定期转储服务**

根据第二章分析，应用程序需要设立后台线程，定期执行命令，以实现定期转储并自动清理过期转储文件。在Blazor应用程序中，需要通过服务实现。先设计服务DumpDbService实现数据库的转储：

public class DumpDbService

{

private readonly ILogger<DumpDbService> \_logger;

public DumpDbService(ILogger<DumpDbService> logger)

{

\_logger = logger;

}

public async Task DumpAsync(string host, string user, string password, string database, string outputFile)

{

try

{

var processStartInfo = new ProcessStartInfo

{

FileName = "cmd.exe",

Arguments = $"/c mysqldump -h {host} -u {user} -p{password} {database} > \"{outputFile}\"",

RedirectStandardError = true,

UseShellExecute = false,

CreateNoWindow = true

};

using var process = new Process

{

StartInfo = processStartInfo

};

process.Start();

await process.WaitForExitAsync();

if (process.ExitCode != 0)

{

var error = await process.StandardError.ReadToEndAsync();

throw new Exception($"Database dump failed: {error}");

}

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error occured in dumping database");

}

}

}

本系统将转储文件存放至Dumps文件夹中。设计服务ClearDumpService清理保存了30分钟以上的转储文件：

public class ClearDumpService

{

public static readonly TimeSpan RetentionDuration = TimeSpan.FromMinutes(30);

private readonly ILogger<ClearDumpService> logger;

public ClearDumpService(ILogger<ClearDumpService> logger)

{

\_logger = logger;

}

public Task ClearOutdateDumps(string directory)

{

var directoryInfo = new DirectoryInfo(directory);

foreach (var file in directoryInfo.GetFiles("\*.sql"))

{

if (DateTime.Now - file.LastWriteTime > RetentionDuration)

{

try

{

file.Delete();

\_logger.LogInformation("Deleted outdate dump file {file}", file);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error occured in clearing outdated dump file");

}

}

}

return Task.CompletedTask;

}

}

最后实现服务BackgroundDumpService实现每隔5分钟自动转储和清理：

public class BackgroundDumpService : BackgroundService

{

public static readonly string DumpDirectory = "Dumps";

public static readonly TimeSpan DumpInterval = TimeSpan.FromMinutes(5);

private readonly IServiceProvider \_serviceProvider;

private readonly ILogger<BackgroundDumpService> \_logger;

public BackgroundDumpService(IServiceProvider serviceProvider, ILogger<BackgroundDumpService> logger)

{

\_serviceProvider = serviceProvider;

\_logger = logger;

}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

{

\_logger.LogInformation("BackgroundDumpService started");

while (!stoppingToken.IsCancellationRequested)

{

try

{

await DumpDb();

await ClearDumps();

await Task.Delay(DumpInterval, stoppingToken);

}

catch (Exception ex)

{

\_logger.LogError(ex, "Error occured in BackgroundDumpService");

}

}

\_logger.LogInformation("BackgroundDumpService exited");

}

protected async Task DumpDb()

{

var timestamp = DateTime.Now;

var dumpService = \_serviceProvider.GetRequiredService<DumpDbService>();

var outputFile = $"Dumps/dump\_{timestamp:yyyyMMdd\_HHmmss}.sql";

await dumpService.DumpAsync("localhost", "root", "123456", "ComponentTradeCenter", outputFile);

\_logger.LogInformation("Database dumped to {OutputFile}", outputFile);

}

protected async Task ClearDumps()

{

var dumpService = \_serviceProvider.GetRequiredService<ClearDumpService>();

await dumpService.ClearOutdateDumps(DumpDirectory);

}

}

在应用程序启动时注册服务：

builder.Services.AddSingleton<DumpDbService>();

builder.Services.AddSingleton<ClearDumpService>();

builder.Services.AddHostedService<BackgroundDumpService>();

服务会在应用程序启动时自动运行，实现定期转储和清理转储文件。

# 界面设计

## 组件设计

基于Blazor架构的应用程序的界面是通过组件构建的。为便于快捷高效地设计网页界面，引用了组件库BootstrapBlazor来构建页面。

根据第一章的分析，本系统需要根据用户的类型和在线情况对数据设置访问权限，因此诸多组件需要注入已注册的服务。为减少重复代码，设计基类OnlineComponent，定义如下：

public class OnlineComponent : ComponentBase

{

[Inject]

[NotNull]

protected CenterDbContext? DbContext { get; set; }

[Inject]

[NotNull]

protected NavigationManager? NavigationManager { get; set; }

[Inject]

[NotNull]

protected OnlineUsersService? OnlineUsersService { get; set; }

[CascadingParameter]

[NotNull]

public MainLayout? MainLayOut { get; set; }

[NotNull]

public UserBase? User { get; set; }

protected override void OnInitialized()

{

base.OnInitialized();

var uri = NavigationManager.ToAbsoluteUri(NavigationManager.Uri);

var queryParams = Microsoft.AspNetCore.WebUtilities.QueryHelpers.ParseQuery(uri.Query);

if (queryParams.TryGetValue("customid", out var customId) &&

DbContext.TryGetUser(customId, out var user) &&

OnlineUsersService.IsOnline(user))

{

User = user;

MainLayOut.User = user;

return;

}

NavigationManager.NavigateTo("/error?info=Not%20Found");

}

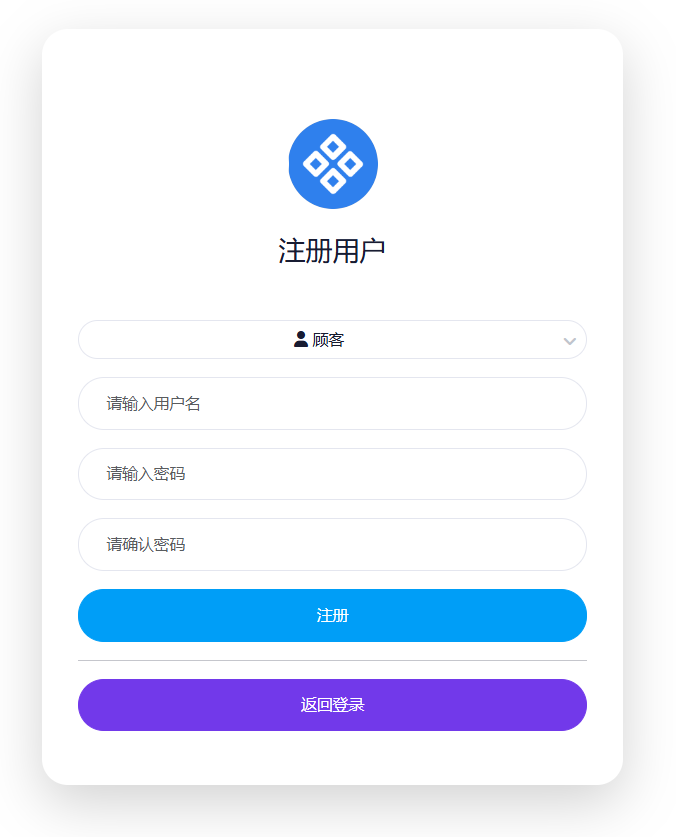
}

本类中User属性表示操作当前组件的用户，组件可以根据该属性调整组件内部属性的可见性；而重载基类函数OnInitialized以实现通过当前的URL来获取用户的ID和类型。

所有需要使用数据库连接、用户在线服务的组件都继承自本类。

## 登录/注册页面

登录和注册是用户使用本系统的第一步，设计安全、高效而美观的登录/注册页面是尤为重要的。本系统将登录和注册设为两个页面，通过URL进行跳转，使用基类的服务实现数据库查询插入和在线用户的管理。

**图4-1　登录页面 图4-2　注册页面**

　　用户登录时，需要向数据库查询当前用户类型下是否存在具有当前用户名且未注销的用户，比对密码是否正确；注册时，需要确认当前用户类型中是否存在相同的用户名且未注销的用户；并且出于对系统安全性的考虑，不允许用户以管理员身份注册账号。

## 个人中心页面

出于对用户体验和系统可维护性的考虑，设计个人中心页面，为用户提供修改用户名、密码、地址和电话等信息的功能，并提供登出和注销账号的账号管理功能。



**图4-3　个人中心页面**

在提交更改时，注意需要验证用户名字段是否唯一或为空；更改的密码字段是否不为空且与确认密码一致。

## 表单页面

表单页面负责从数据库查询并展示数据，新增、修改和删除用户可见的数据，所以表单页面是本系统前端部分的重中之重。本系统使用BootstrapBlazor库中的Table组件，其内置了展示、添加、删除、查询、筛选等数据操作，同时允许设计人员灵活设置功能权限。



**图4-4　零件信息表**



**图4-5　收购信息表（需求表）**



**图4-6　出售信息表（供应表）**



**图4-7　供需关系表**



**图4-8　交易建议表**



**图4-9　在途交易表（协议表）**



**图4-10　交易记录表**



**图4-11　用户信息表**

以不同的身份操作表时，表格的可编辑性、可删除性、可添加性都会随之改变，本系统的表单设计确保了用户对表单数据操作的安全性。

## 选择组件

用户在添加、修改表单数据时，可能需要选择别的表的记录。为在应用程序操作上保证参照完整性，系统为不同的实体各自设计了选择组件。选择组件是对Select组件的包装，以下是选择组件的抽象基类EntitySelectBase<TItem>的定义：

public abstract class EntitySelectBase<TItem>

: ComponentBase where TItem : EntityBase

{

[Inject]

[NotNull]

protected CenterDbContext? DbContext { get; set; }

protected Dictionary<SelectedItem, TItem> Items { get; set; }

[Parameter]

public int Value { get; set; }

[Parameter]

public EventCallback<int> ValueChanged { get; set; }

protected int InternalValue

{

get => Value;

set

{

Value = value;

ValueChanged.InvokeAsync(Value);

}

}

public EntitySelectBase()

{

Items = new Dictionary<SelectedItem, TItem>();

}

protected override void OnInitialized()

{

base.OnInitialized();

}

}

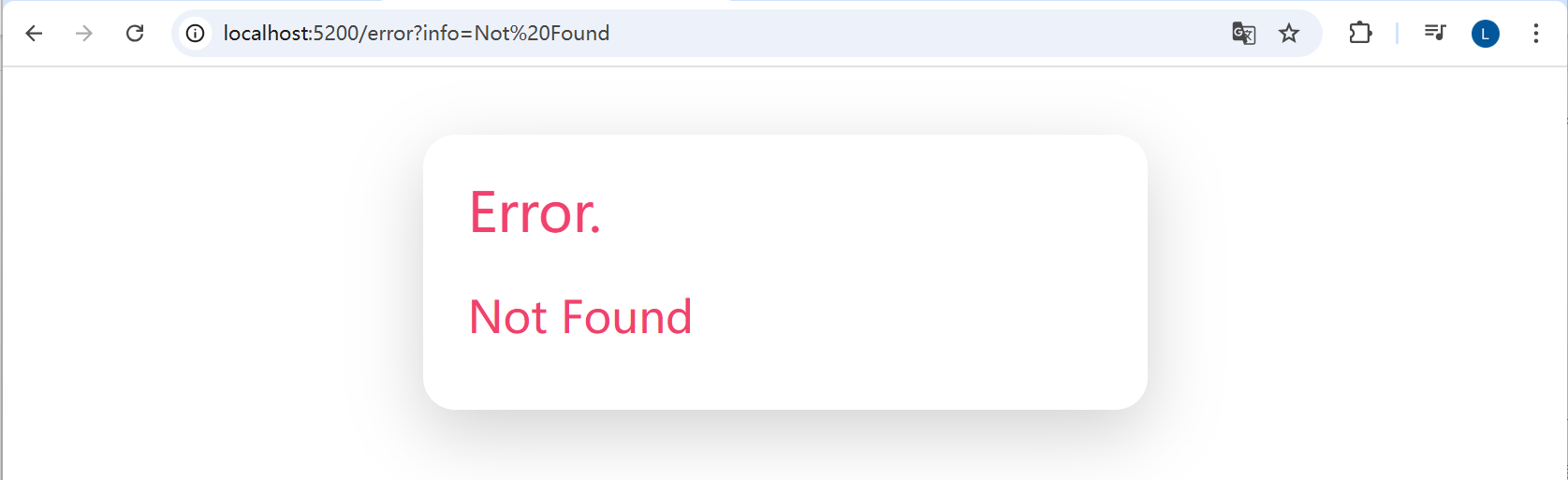
　　由此派生的组件类在初始化时向数据库加载所有未被删除的记录,以加载选择项。以下是部分实体类的选择组件：

**图4-12　零件和需求选择组件**

## 错误页面

当系统发生异常时，需要将异常信息展示在界面上，便于开发人员和用户获知错误原因。因此，设置一个错误页面是必要的。本系统通过URL向页面传入错误信息，在组件中解析信息并展示在前端。设计如下：



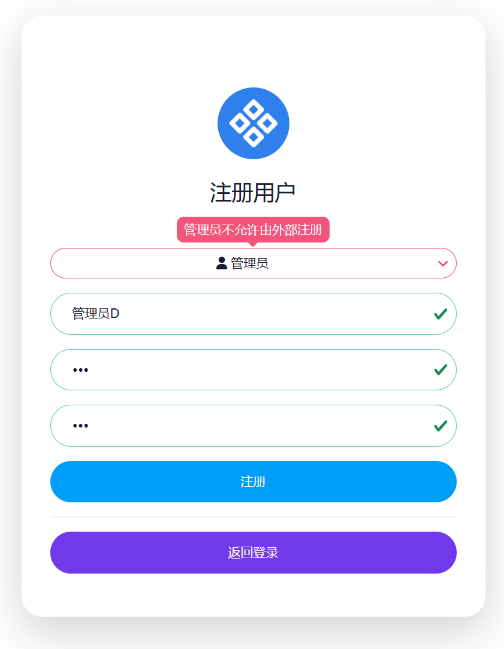
**图4-13　错误页面**

# 系统测试

　　为确保系统设计的正确性和健壮性，对系统的子模块和核心功能进行测试。

## 登录注册测试

已知系统内不存在顾客A、供应商B、交易员C，现尝试以顾客A的身份登录系统，则显示“用户名或密码错误”。结果如图5-1所示。

**图5-1　登录注册子模块测试**

　　以他们的身份注册账号，均能顺利通过注册进入系统。倘若以管理员D的身份注册账号，则显示“管理员不允许由外部注册”。

　　登录/注册子模块按预期工作，通过测试。

## 个人信息测试

以顾客A的身份登录，修改用户名为“顾客7535”，提交更改后显示修改成功；随后将用户名再改回“顾客A”；登出后，再以供应商B的身份登录，修改电话为“10086”：

**图5-2　个人信息子模块测试**

　　个人信息子模块按预期工作，通过测试。

## 模拟交易测试

由顾客A、供应商B和交易员C模拟一场完整的零件交易，以测试系统的交易功能。

1. **顾客发布收购信息：**

顾客A浏览零件信息，发布一条“电机”的收购信息：



**图5-3　顾客发布收购信息**

1. **供应商发布供应信息：**

供应商B浏览零件信息，发布一条“电机”的出售信息：



**图5-4　供应商发布出售信息**

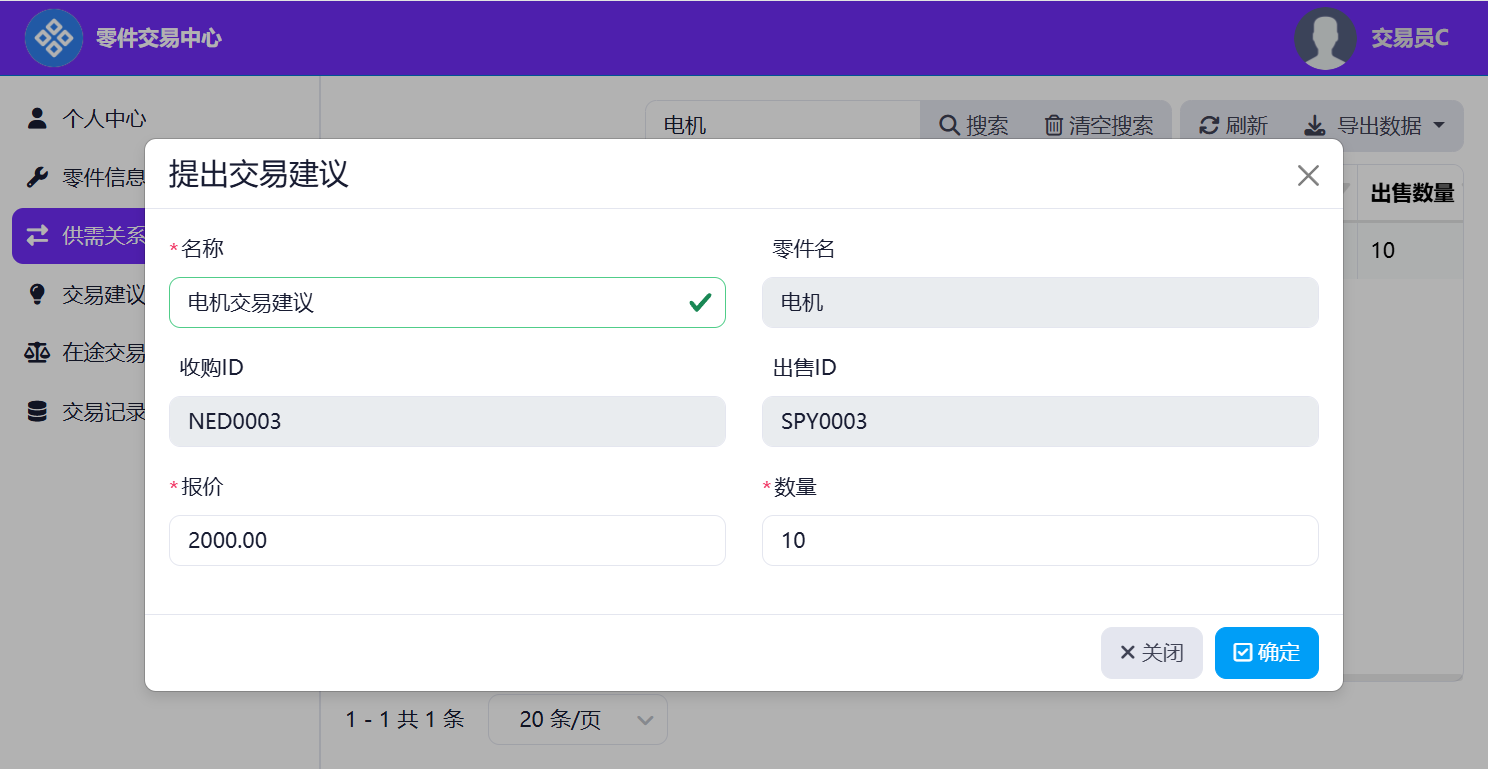
1. **交易员分析供需关系并提出交易建议：**

交易员C浏览供需关系表，搜索“电机”，能查询到顾客A和供应商B的供需信息：



**图5-5　交易员分析供需关系**

交易员C对顾客A和供应商B发布的收购和出售信息提出交易建议：



**图5-6　交易员向双方提出交易建议**

1. **顾客和供应商处理交易建议：**

顾客A和供应商B接受交易员C提出的建议：



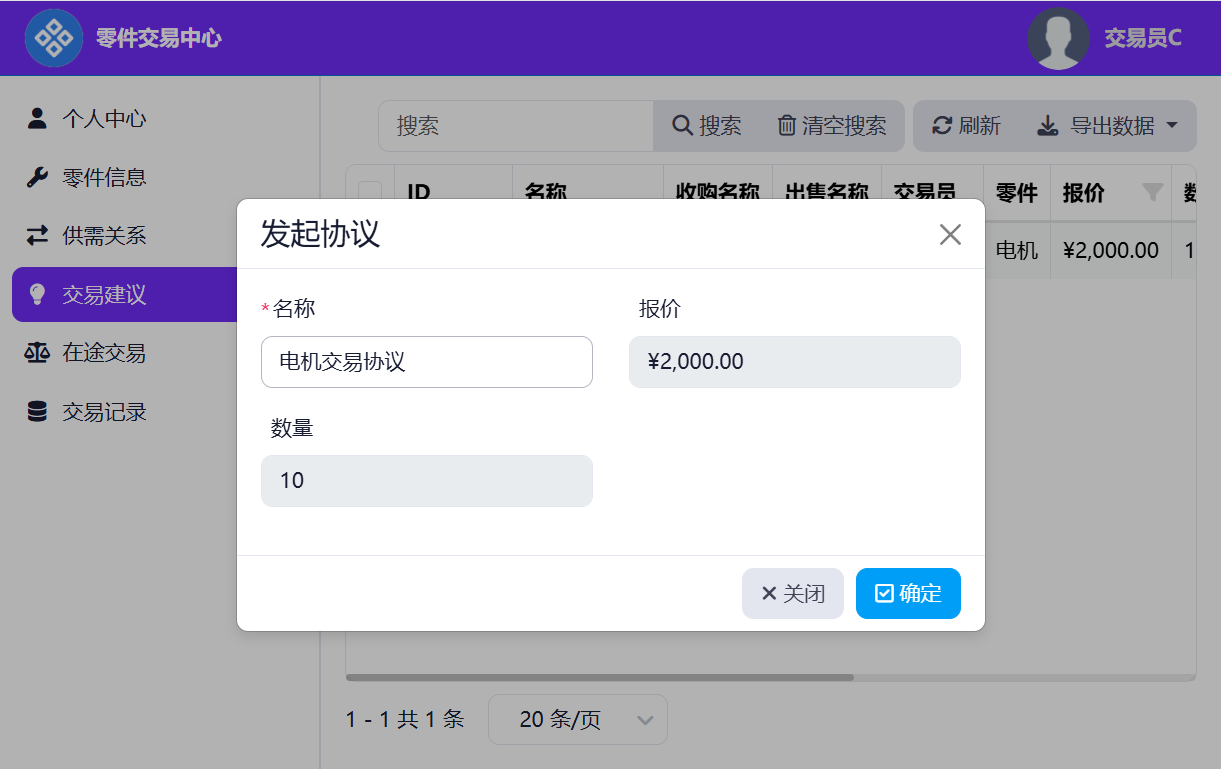
**图5-7　顾客同意交易建议**



**图5-8　供应商同意交易建议**

1. **交易员发起交易协议：**

交易员C根据顾客A和供应商B对建议的意向，发起交易协议：



**图5-9　交易员发起交易协议**

1. **顾客和供应商上传签名：**

顾客A和供应商B根据交易协议，上传各自的签名文件：



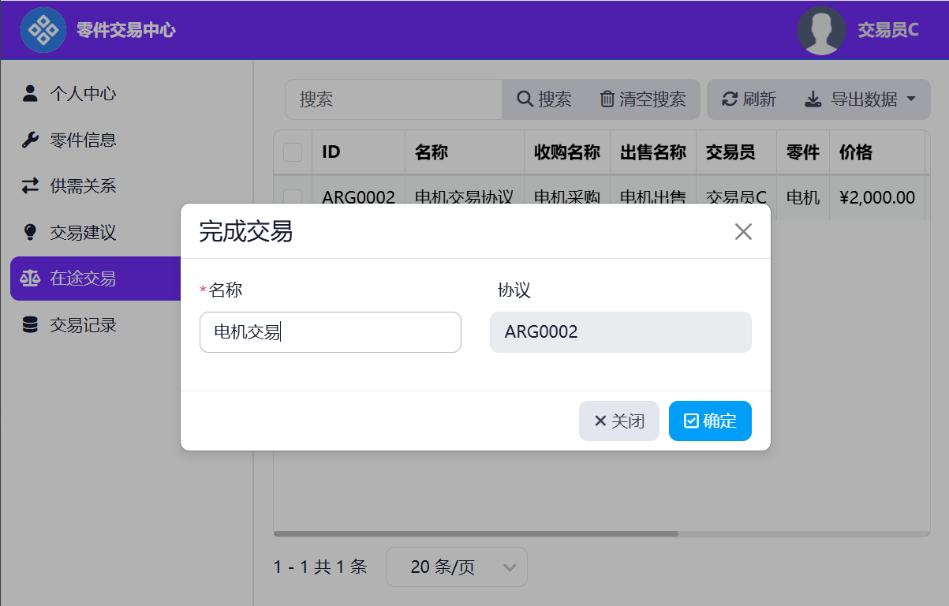
**图5-10　顾客上传签名**



**图5-11　供应商上传签名**

1. **交易员审批协议：**

交易员C审查协议是否达成，通过协议审批，交易完成，并生成交易记录：



**图5-12　交易员审批协议**



**图5-13　生成交易记录**

零件交易核心功能按预期工作，通过测试。

## 删改测试

由于系统使用标记删除时间实现软删除，并使用了两个触发器实现用户注销时关联的供需信息的选择性级联删除。该功能的测试需要注销顾客A，使用Navicat观察数据库数据：

**图5-14　交易记录**

可以发现删除时间被同时标记，软删除功能通过测试。

管理员被赋予修改所有表数据的权限，因此还需要测试表单数据的修改功能。这里将交易建议名称为“电机交易建议”的交易员更改为“trad”，并修改供应商的交易意向为“拒绝”：



**图5-15　修改交易建议界面**



**图5-16　交易建议修改结果**

查询数据观察到修改结果正确，测试通过。

# 总结与心得

通过本次零件交易中心数据库系统的开发，我深刻体会到数据库技术在现代信息化管理中的重要作用。从需求分析到系统设计，再到数据库的实现与测试，每一个环节都让我认识到开发工作中条理性和逻辑性的重要性。在需求调研阶段，我学会了如何与业务场景结合，提炼出关键功能需求；在系统设计过程中，合理的数据库表结构设计不仅直接影响系统性能，也为后续功能实现奠定了基础。

在开发的实际操作中，我遇到了许多挑战，例如如何优化查询语句、如何处理并发事务以及如何保证数据的一致性与完整性。通过查阅资料和团队协作，这些问题逐步得到了解决，也让我积累了不少实战经验。这让我更加明白，理论知识只有在实际应用中才能发挥最大价值。

总体而言，这次课设不仅让我巩固了数据库开发的知识，也让我对软件开发的全过程有了更深入的理解。这将是我未来学习和实践的重要起点。

# 参考资料

1. 王珊, 萨师煊. 数据库系统概论（第五版） [M]. 北京: 高等教育出版社, 2014.
2. Microsoft. Microsoft Learn: ASP.NET Core Blazor [EB/OL]. [2024-12-09]. https://learn.microsoft.com/zh-cn/aspnet/core/blazor/?view=aspnetcore-8.0.
3. BootstrapBlazor. BootstrapBlazor 官方文档 [EB/OL]. [2024-12-09]. <https://www.blazor.zone/components>.