# 《数据库系统和信息管理》课程报告

**课题名称： POS(Point Of Sale) System Project**

课题负责人： 龙家可

学号： 2021141461092

指导教师： 伍元凯

**评阅成绩：**

评阅意见：

提交报告时间：2023 年 6月15日

**数据库驱动的基于Web前端的销售点系统**

软件工程 专业

**学生** XXX **指导老师** 伍元凯

介绍

个体营业户进行销售贩卖的情况在我国非常广泛，尤其是小商贩们，他们是社区和城市中不可或缺的一部分。传统上，这些个体营业户使用简单的记录方法来管理他们的销售活动，这包括手写记录和简单的计算方式。然而，随着科技的进步和环境的变化，他们需要一个更现代化的解决方案来简化销售流程并提高效率。

为了满足小个体营业户的需求，我们开发了一个POS系统，旨在为雇员提供一个简单易用的销售管理工具。该系统的前端使用Web技术（HTML、CSS和JavaScript）实现，具有直观友好的用户界面，使得个体营业户可以轻松上手并进行销售操作，满足他们在销售记录与结算、库存管理方面的需求。

此外， POS系统中数据模式的设计，使得原始数据的统计和进一步处理变得更加方便，帮助他们能够更好地了解销售情况，为未来的决策提供依据。

通过我们开发的POS系统，小个体营业户雇员能够摆脱传统的记账方式，拥有一个现代化、高效和可靠的销售管理工具，改善工作条件，提升经营质量。

目录

[《数据库系统和信息管理》课程报告 0](#_Toc137766972)

[介绍 1](#_Toc137766973)

[需求分析 3](#_Toc137766974)

[ER建模 5](#_Toc137766975)

[关系分析 7](#_Toc137766976)

[SQL建表语句 11](#_Toc137766996)

[具体完整实现 14](#_Toc137767005)

[ 部署图 14](#_Toc137767006)

[ 组件图 14](#_Toc137767007)

[ 分层数据流图 15](#_Toc137767008)

[ 领域类 19](#_Toc137767009)

[ 时序图 25](#_Toc137767010)

[ 数据库驱动细节 26](#_Toc137767011)

[ 前端实现 32](#_Toc137767012)

[总结 42](#_Toc137767013)

需求分析

在项目需求分析阶段，根据对一般销售过程的调查抽象，得出了以下需求：

1. Customers (顾客):

当顾客来到销售点时，他们可以通过系统进行结账操作。

* 1. 系统应支持顾客会员信息的录入，包括姓名、联系方式和地址。
  2. 对于新会员，系统可以录入相关信息。
  3. 系统应能够根据顾客的会员账号进行顾客的识别和管理。

1. Orders (订单):

销售员根据顾客需求创建填写订单信息，生成订单号完成订单存储。

* 1. 系统应具备创建订单的功能，包括关联顾客信息、订单日期和总金额。
  2. 每个订单可以包含多个商品，需要记录商品的数量、单价和总价。

1. Products (商品):

系统应该能够对商品的信息进行管理，确保商品供应和库存准确性。

* 1. 系统应支持商品信息的录入，包括商品名称、价格和库存数量。
  2. 商品应具备唯一的标识号（ProductID）用于在系统中进行识别和管理。

1. Payments (支付):

在订单提交后，顾客进行了有效支付，得到销售员确认后生成。

* 1. 系统需要记录支付相关的信息，包括支付日期、支付方式和支付金额。
  2. 每个支付操作应与对应的订单关联起来。

1. Transaction Log (交易日志):

在Payments (支付)记录生成的同时一齐生成。

* 1. 系统应具备记录交易行为的功能，包括交易日期、顾客信息、销售员信息和交易类型等。
  2. 每次交易操作应该被记录下来，以便后续跟踪和审计。
  3. 交易日志可以与支付记录（Payments）进行验证和关联，以确保交易的准确性和完整性。

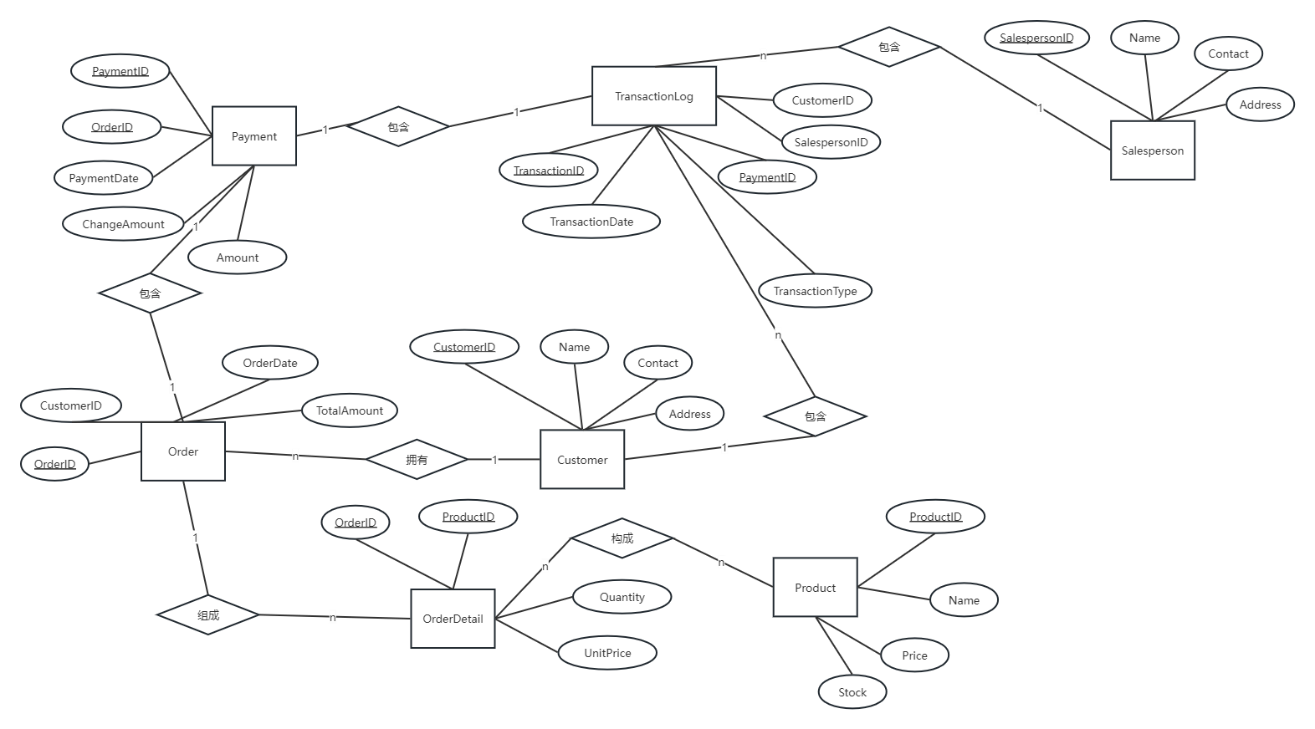
1. Salespersons (销售员):

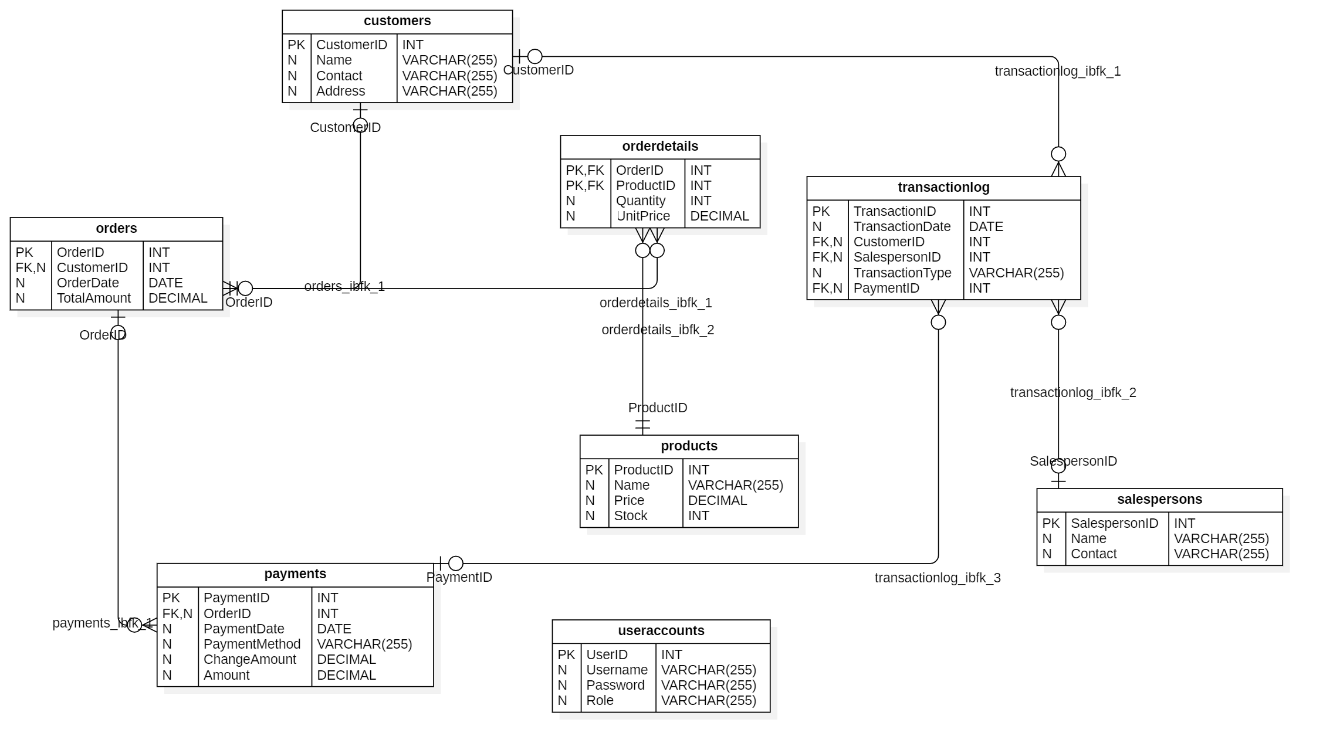
销售员是销售部分的操作者，他们可以使用终端账户进行销售操作。

* 1. 系统应支持销售存储员信息，包括姓名、联系方式和地址。
  2. 销售员可以与订单进行关联，记录销售员的信息以便追踪销售情况。

1. User Accounts (用户账户):
   1. 系统应具备用户账户管理功能，用于描述终端用户。
   2. 每个用户账户应包含用户名、密码和角色信息。
   3. 用户账户的目的是区分和管理不同的终端设备权限，而非使用者。
      1. 多个销售员可以在一台设备上共享同一个用户账户，但在进行销售操作时需要先设定当前操作者的工号。

ER建模

* ER图（user accounts略）
* UML图



1. 顾客（customers）和订单（orders）之间的关系是一对多（One-to-Many）。一个顾客可以拥有多个订单。在订单表中，通过CustomerID字段与顾客表建立外键关系。
2. 订单（orders）和支付（payments）之间的关系是一对一（One-to-One）。一个订单对应一个支付记录。在支付表中，通过OrderID字段与订单表建立外键关系。
3. 订单（orders）和订单详情（orderdetails）之间的关系是一对多（One-to-Many）。一个订单可以包含多个订单详情。在订单详情表中，通过OrderID字段与订单表建立外键关系。
4. 订单详情（orderdetails）和产品（products）之间的关系是多对多（Many-to-Many）。多个订单详情对应多个产品。在订单详情表中，通过ProductID字段与产品表建立外键关系。
5. 交易日志（transactionlog）与顾客（customers）、销售员（salespersons）之间的关系是多对一（Many-to-One），和支付（payments）之间的关系是一对一。一个交易日志记录对应一个顾客、一个销售员和一个支付记录。在交易日志表中，通过CustomerID、SalespersonID和PaymentID字段与对应的表建立外键关系。

关系分析

* **关系：**

**Table: customers**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| CustomerID | int | 顾客ID |
| Name | varchar | 姓名 |
| Contact | varchar | 联系方式 |
| Address | varchar | 地址 |

**主键 (Primary Key): CustomerID**

**Table: orders**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| OrderID | int | 订单ID |
| CustomerID | int | 顾客ID |
| OrderDate | date | 订单日期 |
| TotalAmount | decimal | 订单总金额 |

**主键 (Primary Key): OrderID**

**外键 (Foreign Key): CustomerID refers to customers(CustomerID)**

**Table: orderdetails**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| OrderID | int | 订单ID |
| ProductID | int | 产品ID |
| Quantity | int | 数量 |
| UnitPrice | decimal | 单价 |

**主键 (Primary Key): OrderID, ProductID**

**外键 (Foreign Key): OrderID refers to orders(OrderID)**

**外键 (Foreign Key): ProductID refers to products(ProductID)**

**Table: products**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| ProductID | int | 产品ID |
| Name | varchar | 产品名称 |
| Price | decimal | 价格 |
| Stock | int | 库存数量 |

**主键 (Primary Key): ProductID**

**Table: salespersons**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| SalespersonID | int | 销售员ID |
| Name | varchar | 姓名 |
| Contact | varchar | 联系方式 |
| Address | varchar | 地址 |

**主键 (Primary Key): SalespersonID**

**Table: transactionlog**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| TransactionID | int | 交易记录ID |
| TransactionDate | date | 交易日期 |
| CustomerID | int | 顾客ID |
| SalespersonID | int | 销售员ID |
| TransactionType | varchar | 交易类型 |
| PaymentID | int | 支付ID |

**主键 (Primary Key): TransactionID**

**外键 (Foreign Key): CustomerID refers to customers(CustomerID)**

**外键 (Foreign Key): SalespersonID refers to salespersons(SalespersonID)**

**外键 (Foreign Key): PaymentID refers to payments(PaymentID)**

**Table: payments**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| PaymentID | int | 支付ID |
| OrderID | int | 订单ID |
| PaymentDate | date | 支付日期 |
| PaymentMethod | varchar | 支付方式 |
| ChangeAmount | decimal | 找零金额 |
| Amount | decimal | 支付金额 |

**主键 (Primary Key): PaymentID**

**外键 (Foreign Key): OrderID refers to orders(OrderID)**

**Table: useraccounts**

**Useraccounts**

| **字段名** | **数据类型** | **约束** |
| --- | --- | --- |
| **UserID** | **int** | **主键** |
| **AccountName** | **varchar(255)** |  |
| **Password** | **varchar(255)** |  |
| **Role** | **varchar(255)** |  |

**主键 (Primary Key): UserID**

* **函数依赖与范式分析**

Table: customers

CustomerID -> Name, Contact, Address

这个函数依赖满足BCNF和第三范式要求。属性 Name, Contact, Address 完全依赖于候选键 CustomerID，没有冗余的依赖。

Table: orders

OrderID -> CustomerID, OrderDate, TotalAmount

这个函数依赖满足BCNF和第三范式要求。属性 CustomerID, OrderDate, TotalAmount 完全依赖于候选键 OrderID，没有冗余的依赖。



**Table: orderdetails**

OrderID -> ProductID, Quantity, UnitPrice

这个函数依赖满足BCNF和第三范式要求。属性 ProductID, Quantity, TotalAmount 完全依赖于候选键 OrderID，没有冗余的依赖。

Table: payments

PaymentID -> OrderID, PaymentDate, PaymentMethod, ChangeAmount, Amount

这个函数依赖不满足BCNF和第三范式要求。属性 Amount 可依赖于候选键 OrderID。然而在系统设计中仍然如此保留，是为了满足交易实体和订单实体的完整性。

Table: products

ProductID -> Name, Price, Stock

这个函数依赖满足BCNF和第三范式要求。属性 Name, Price, Stock 完全依赖于候选键 ProductID，没有冗余的依赖。

Table: salespersons

SalespersonID -> Name, Contact, Address

这个函数依赖满足BCNF和第三范式要求。属性 Name, Contact, Address 完全依赖于候选键 SalespersonID，没有冗余的依赖。

Table: transactionlog

TransactionID -> TransactionDate, CustomerID, SalespersonID, TransactionType, PaymentID

这个函数依赖满足BCNF和第三范式要求。属性 TransactionDate, CustomerID, SalespersonID, TransactionType, PaymentID 完全依赖于候选键 TransactionID，没有冗余的依赖。

Table: useraccounts

**UserID -> AccountName, Password, Role**

这个函数依赖满足BCNF和第三范式要求。属性 AccountName, Password, Role完全依赖于候选键 UserID，没有冗余的依赖。

SQL建表语句

## Table structure for table customers

CREATE TABLE `customers` (  
 `CustomerID` int NOT NULL AUTO\_INCREMENT,  
 `Name` varchar(255) DEFAULT NULL,  
 `Contact` varchar(255) DEFAULT NULL,  
 `Address` varchar(255) DEFAULT NULL,  
 PRIMARY KEY (`CustomerID`)  
);

## Table structure for table orderdetails

CREATE TABLE `orderdetails` (  
 `OrderID` int NOT NULL,  
 `ProductID` int NOT NULL,  
 `Quantity` int DEFAULT NULL,  
 `UnitPrice` decimal(10,2) DEFAULT NULL,  
 PRIMARY KEY (`OrderID`,`ProductID`),  
 KEY `orderdetails\_ibfk\_2` (`ProductID`),  
 CONSTRAINT `orderdetails\_ibfk\_1` FOREIGN KEY (`OrderID`) REFERENCES `orders` (`OrderID`),  
 CONSTRAINT `orderdetails\_ibfk\_2` FOREIGN KEY (`ProductID`) REFERENCES `products` (`ProductID`)  
);

## Table structure for table orders

CREATE TABLE `orders` (  
 `OrderID` int NOT NULL AUTO\_INCREMENT,  
 `CustomerID` int DEFAULT NULL,  
 `OrderDate` date DEFAULT NULL,  
 `TotalAmount` decimal(10,2) DEFAULT NULL,  
 PRIMARY KEY (`OrderID`),  
 KEY `orders\_ibfk\_1` (`CustomerID`),  
 CONSTRAINT `orders\_ibfk\_1` FOREIGN KEY (`CustomerID`) REFERENCES `customers` (`CustomerID`)  
);

## Table structure for table payments

CREATE TABLE `payments` (  
 `PaymentID` int NOT NULL AUTO\_INCREMENT,  
 `OrderID` int DEFAULT NULL,  
 `PaymentDate` date DEFAULT NULL,  
 `PaymentMethod` varchar(255) DEFAULT NULL,  
 `ChangeAmount` decimal(10,2) DEFAULT NULL,  
 `Amount` decimal(10,2) DEFAULT NULL,  
 PRIMARY KEY (`PaymentID`),  
 KEY `payments\_ibfk\_1` (`OrderID`),  
 CONSTRAINT `payments\_ibfk\_1` FOREIGN KEY (`OrderID`) REFERENCES `orders` (`OrderID`)  
) ;

## Table structure for table products

CREATE TABLE `products` (  
 `ProductID` int NOT NULL AUTO\_INCREMENT,  
 `Name` varchar(255) DEFAULT NULL,  
 `Price` decimal(10,2) DEFAULT NULL,  
 `Stock` int DEFAULT NULL,  
 PRIMARY KEY (`ProductID`)  
);

## Table structure for table salespersons

CREATE TABLE `salespersons` (  
 `SalespersonID` int NOT NULL AUTO\_INCREMENT,  
 `Name` varchar(255) DEFAULT NULL,  
 `Contact` varchar(255) DEFAULT NULL,  
 `Address` varchar(255) DEFAULT NULL,  
 PRIMARY KEY (`SalespersonID`)  
) ;

## Table structure for table transactionlog

CREATE TABLE `transactionlog` (  
 `TransactionID` int NOT NULL AUTO\_INCREMENT,  
 `TransactionDate` date DEFAULT NULL,  
 `CustomerID` int DEFAULT NULL,  
 `SalespersonID` int DEFAULT NULL,  
 `TransactionType` varchar(255) DEFAULT NULL,  
 `PaymentID` int DEFAULT NULL,  
 PRIMARY KEY (`TransactionID`),  
 KEY `transactionlog\_ibfk\_1` (`CustomerID`),  
 KEY `transactionlog\_ibfk\_3` (`PaymentID`),  
 KEY `transactionlog\_ibfk\_2` (`SalespersonID`),  
 CONSTRAINT `transactionlog\_ibfk\_1` FOREIGN KEY (`CustomerID`) REFERENCES `customers` (`CustomerID`),  
 CONSTRAINT `transactionlog\_ibfk\_2` FOREIGN KEY (`SalespersonID`) REFERENCES `salespersons` (`SalespersonID`),  
 CONSTRAINT `transactionlog\_ibfk\_3` FOREIGN KEY (`PaymentID`) REFERENCES `payments` (`PaymentID`)  
);

## -- Table structure for table useraccounts

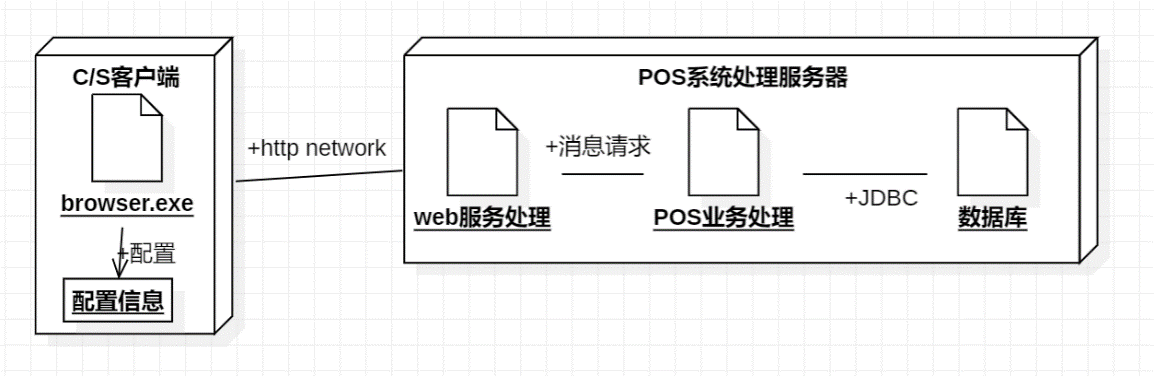
CREATE TABLE `useraccounts` (  
 `UserID` int NOT NULL AUTO\_INCREMENT,  
 `Username` varchar(255) DEFAULT NULL,  
 `Password` varchar(255) DEFAULT NULL,  
 `Role` varchar(255) DEFAULT NULL,  
 PRIMARY KEY (`UserID`)  
);

具体完整实现

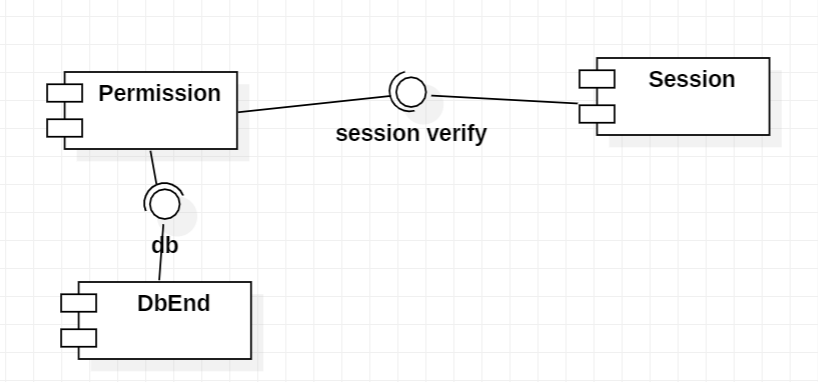
后端语言：Java (环境jdk-17.0.3.7-openj9), 项目依赖：JDBC, Spark, GJson

前端：HTML, CSS(Bootstrap used), JavaScript(jQuery used)

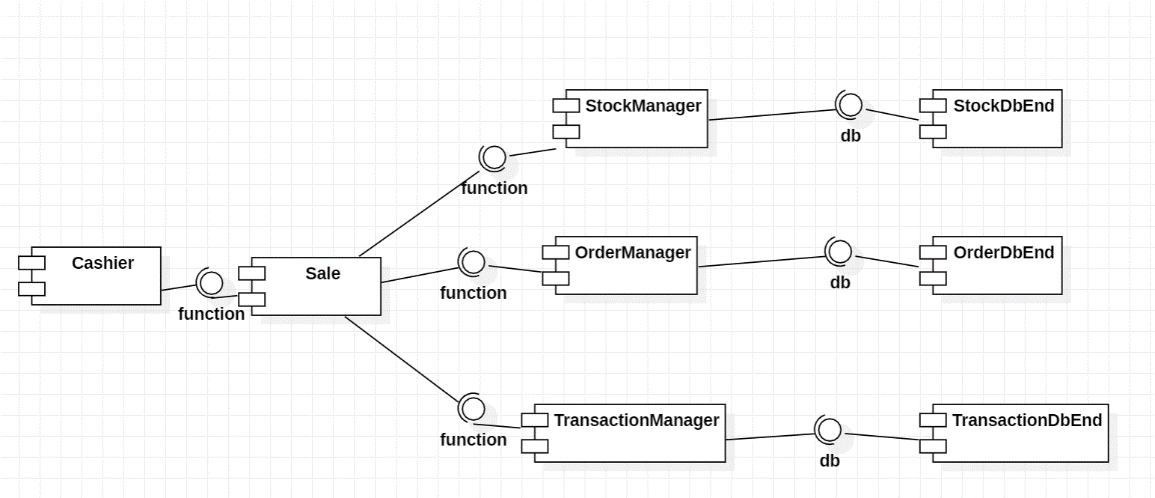
* 部署图

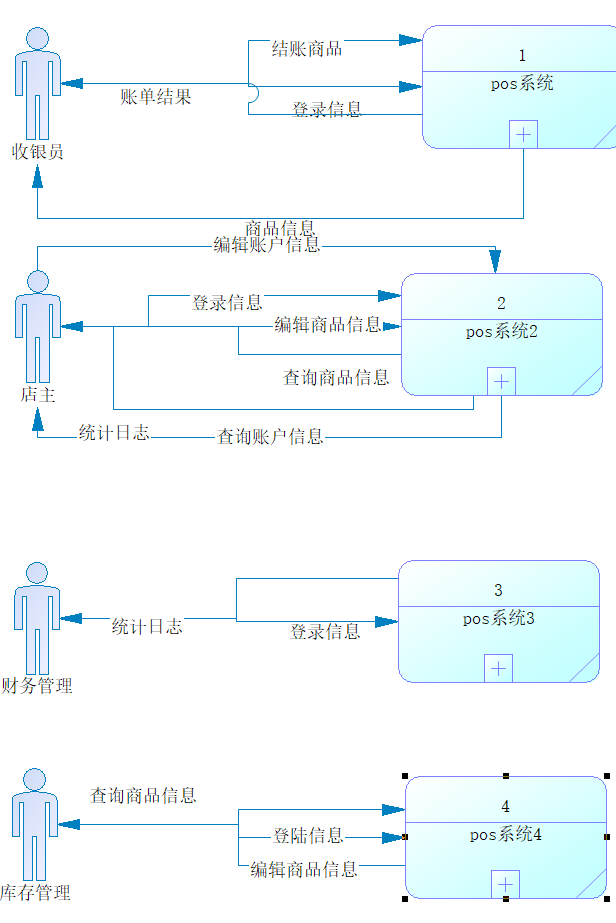


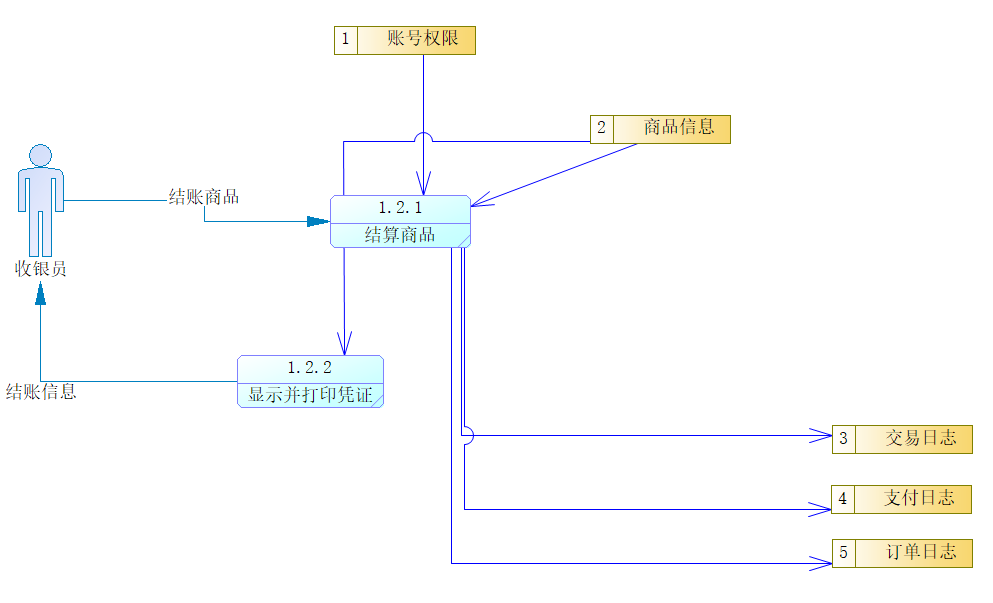
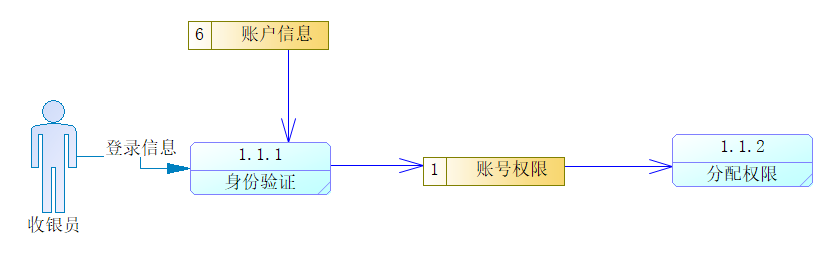
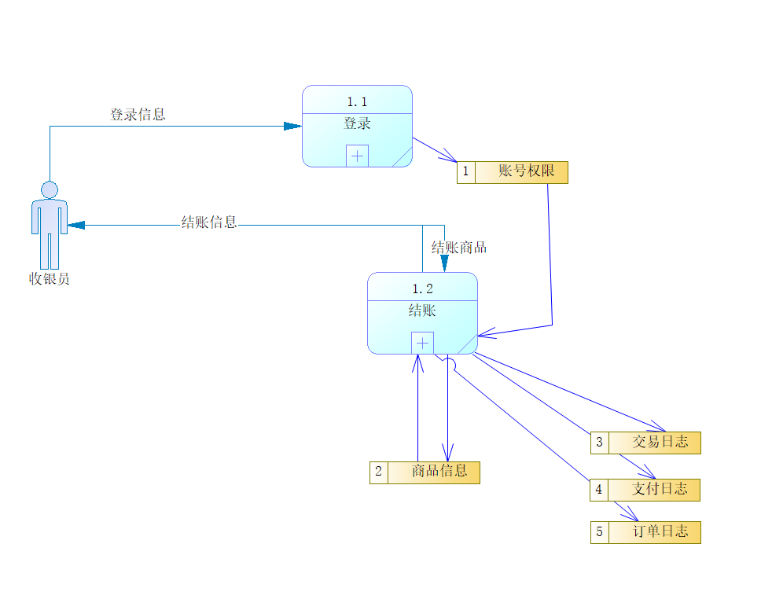
* 组件图
  + 登录部分：

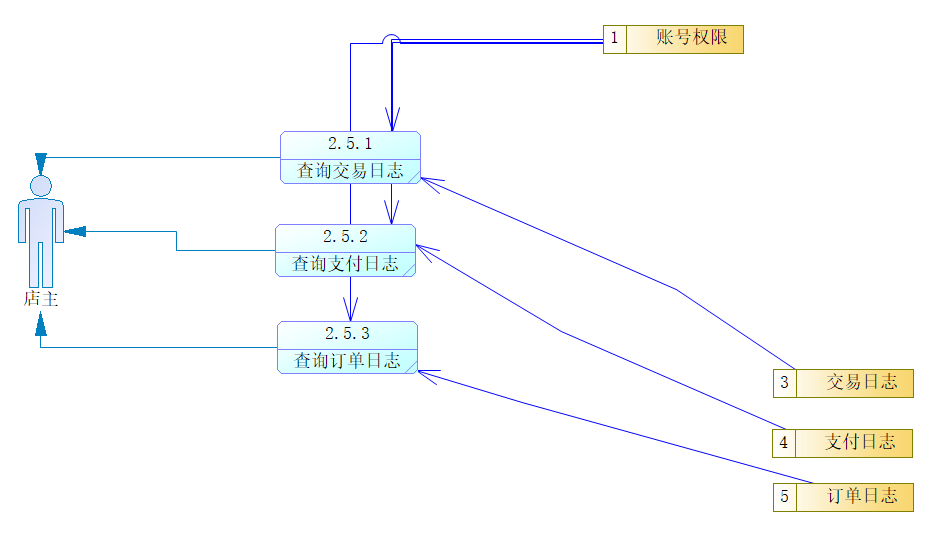
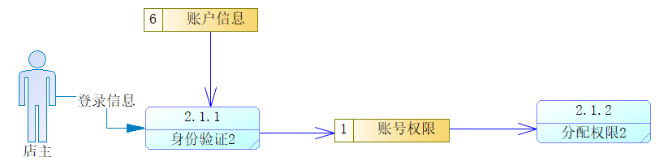
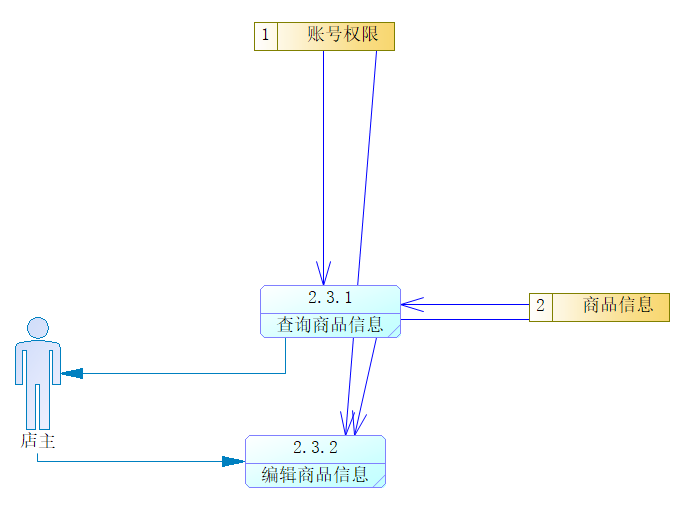
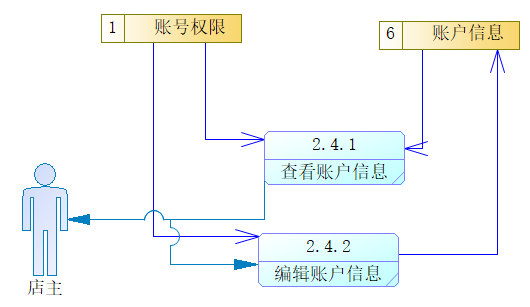
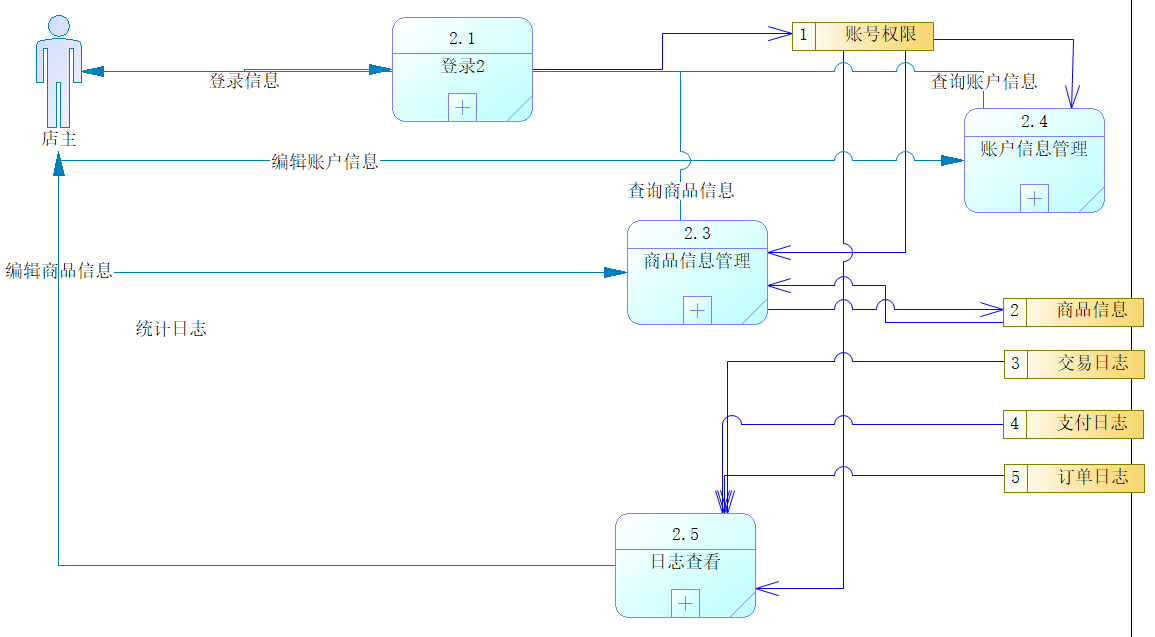


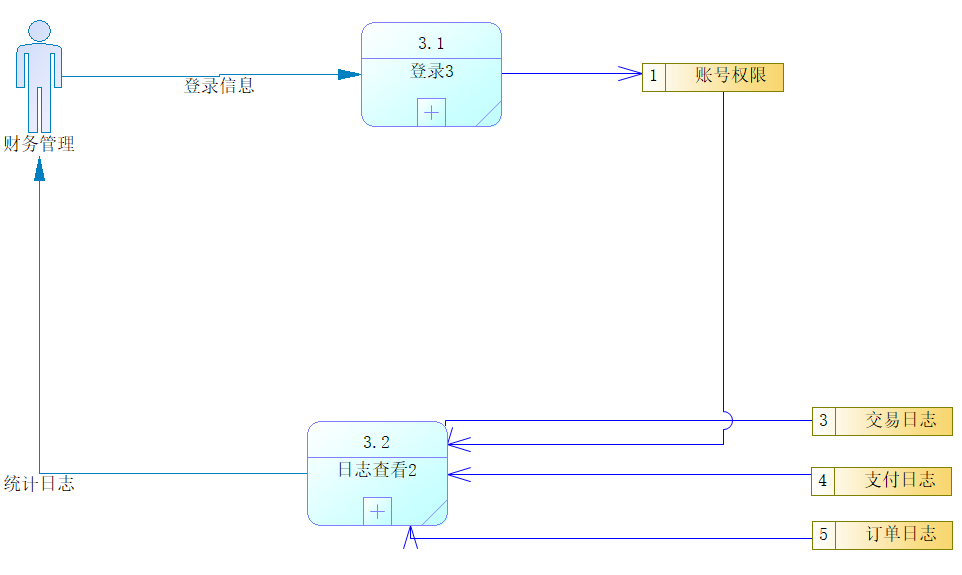
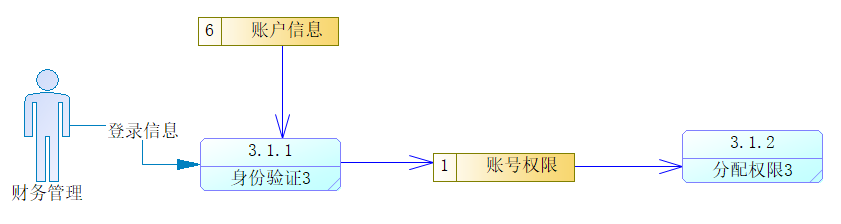
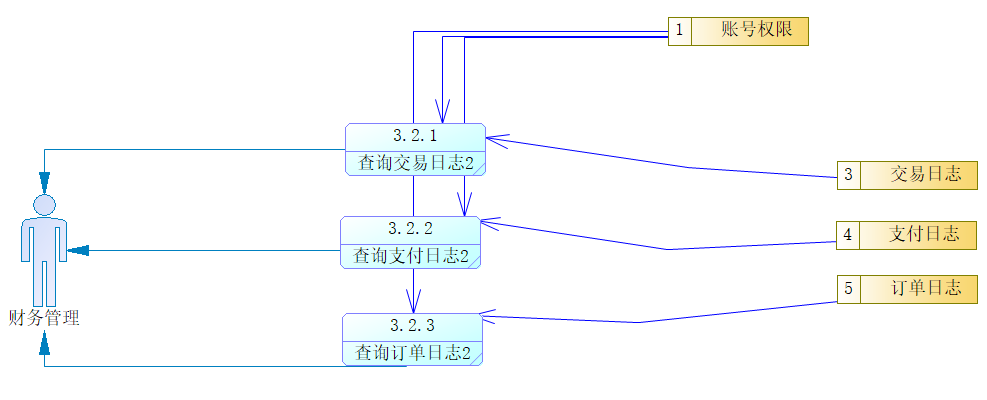
* + 收银部分：

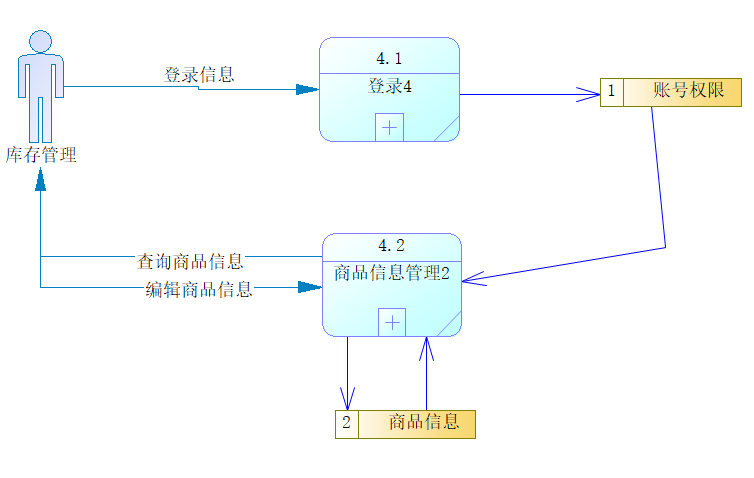
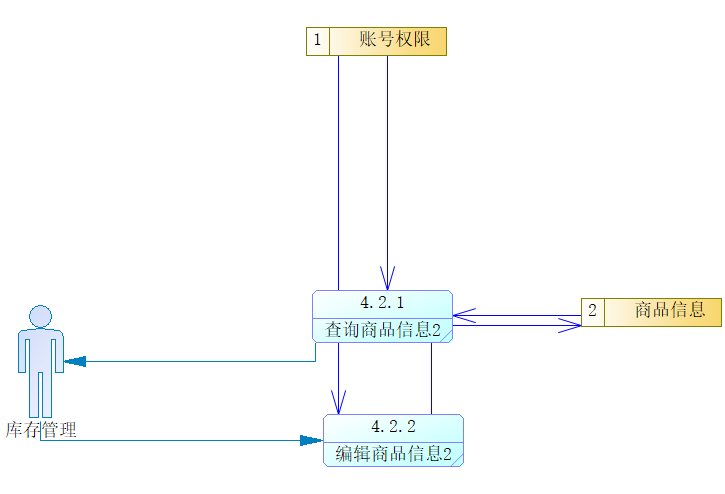


* 分层数据流图







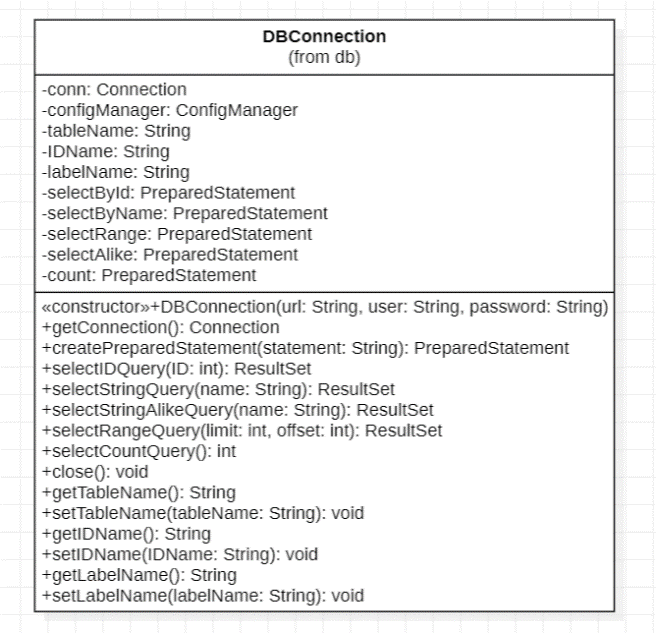


* 领域类

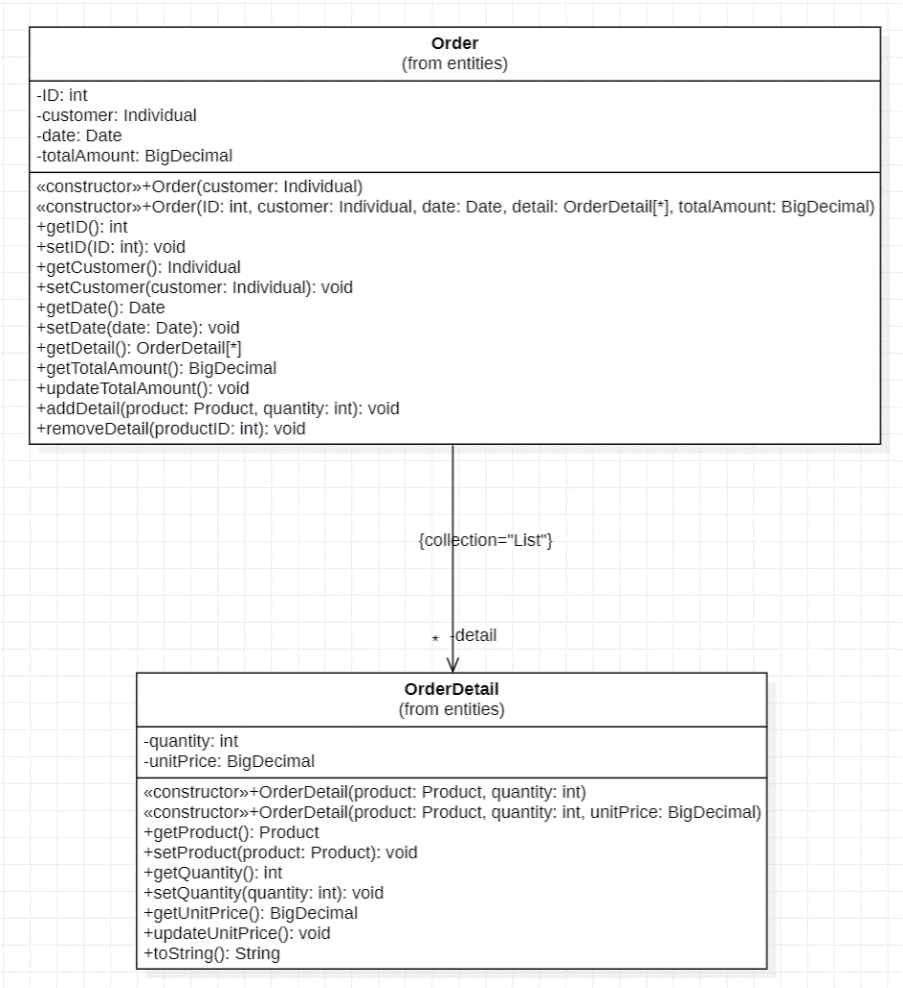
完整类图：

UML类图（数据库业务处理部分）：

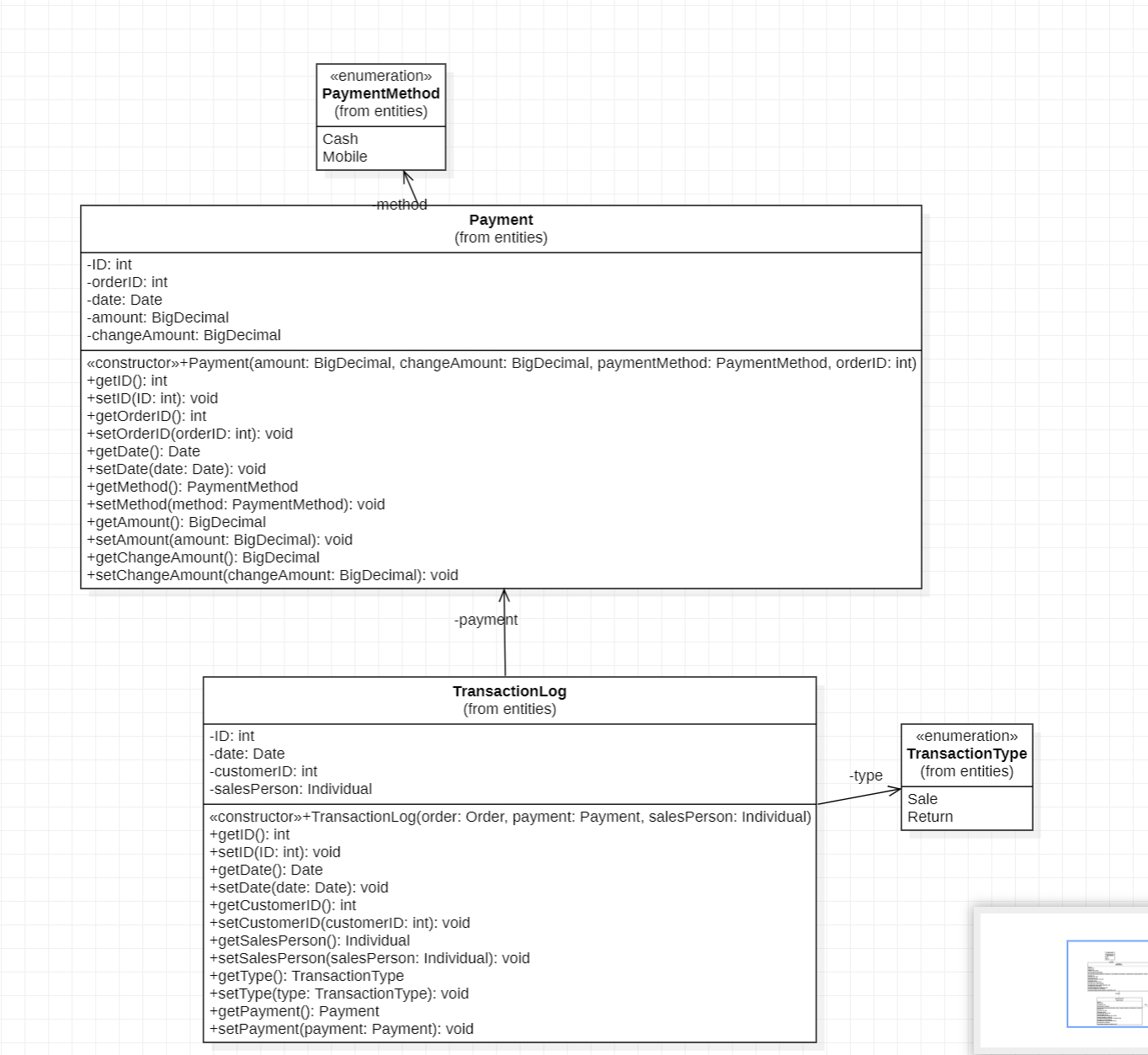
数据库工具类：



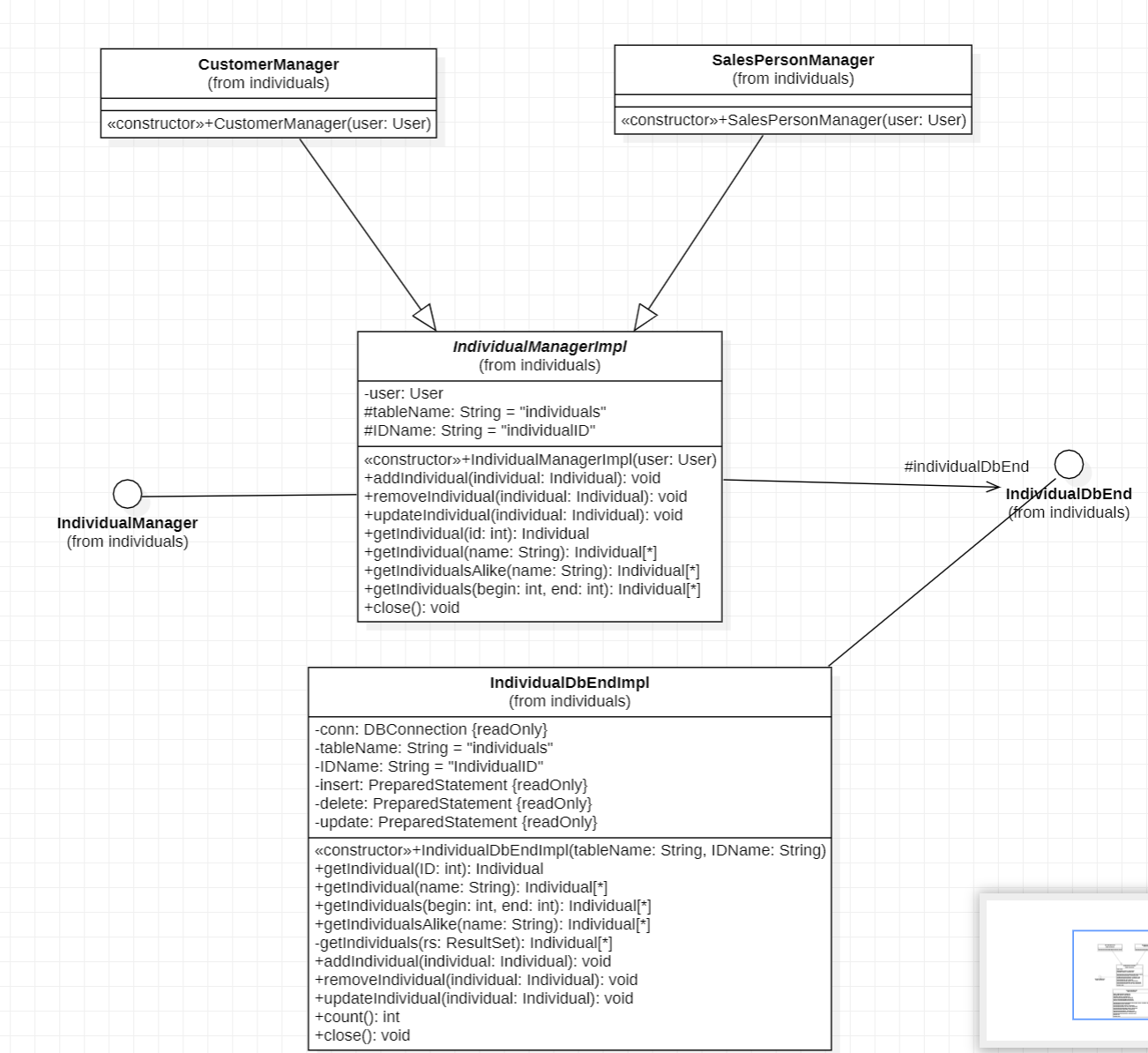
订单类



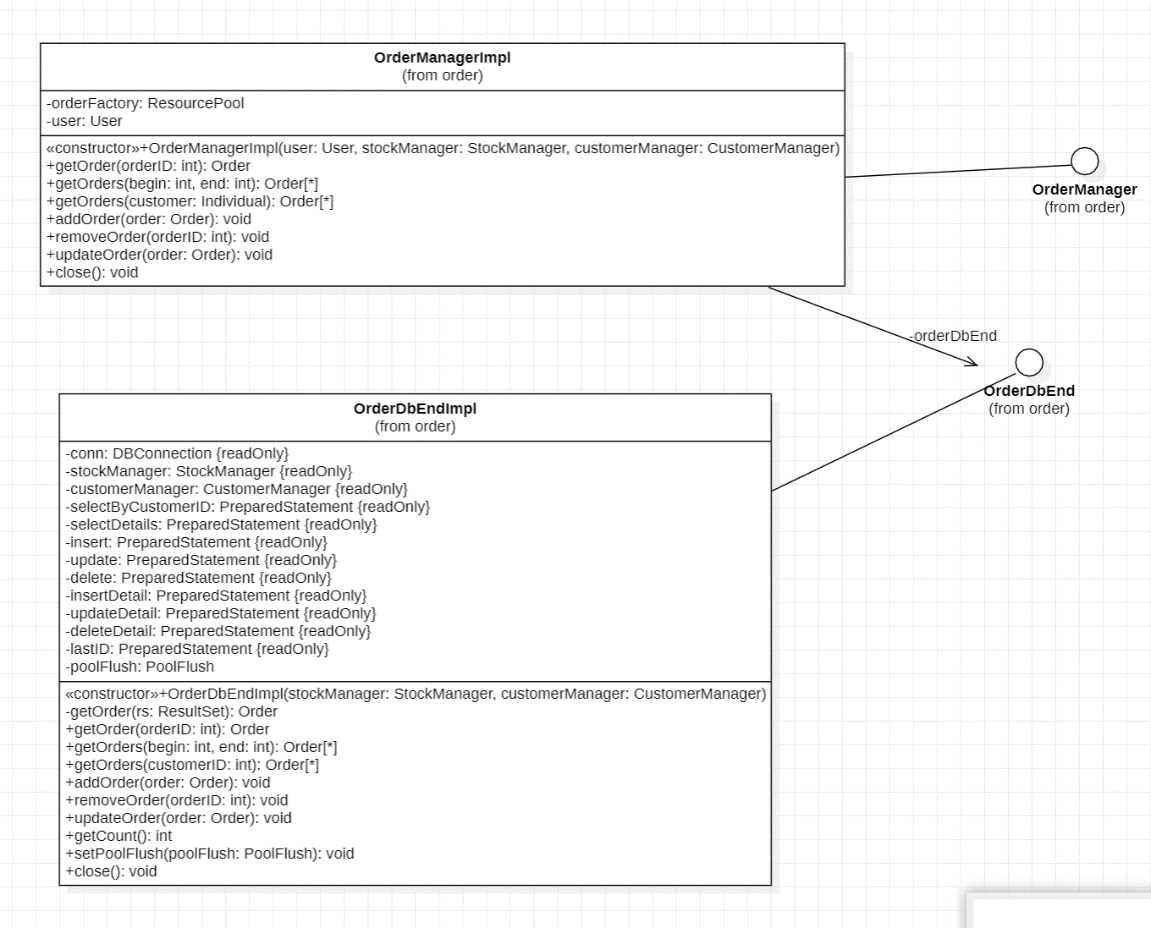
交易信息类：



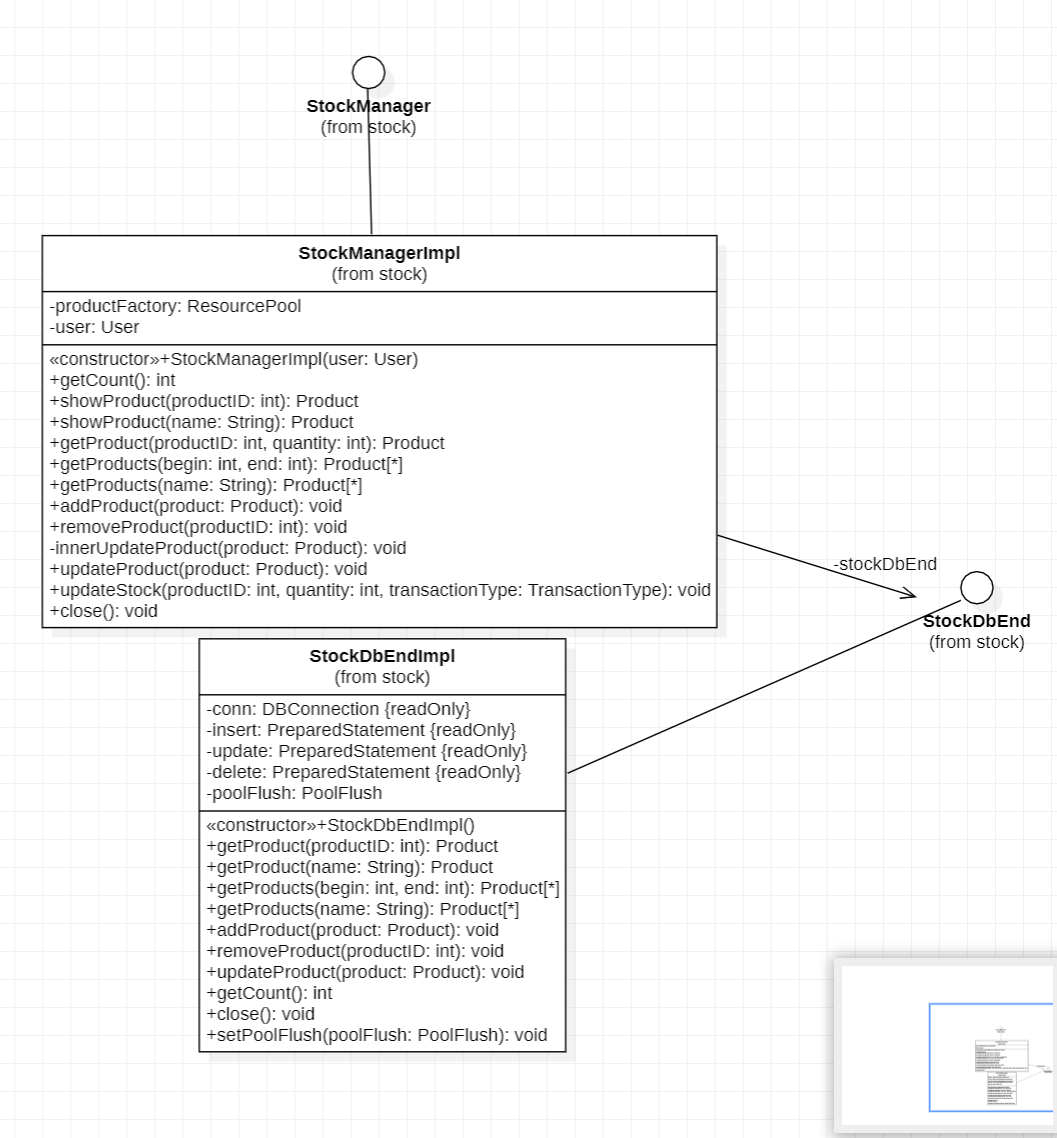
个人信息管理类：



订单管理类：

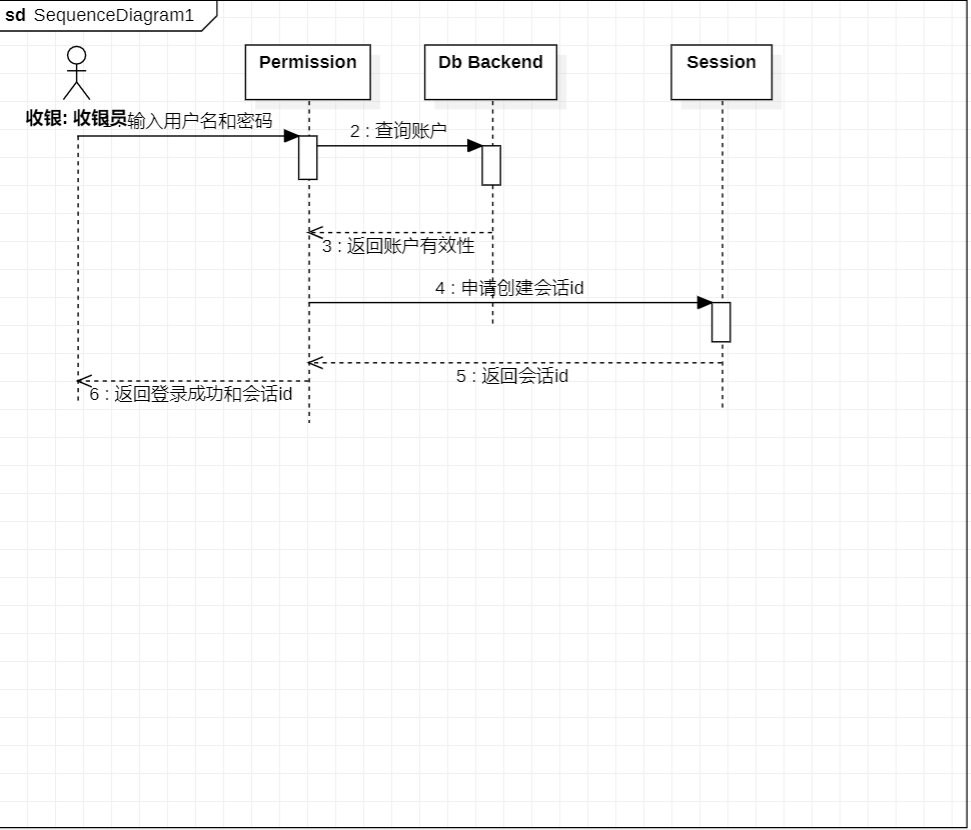


库存管理类：

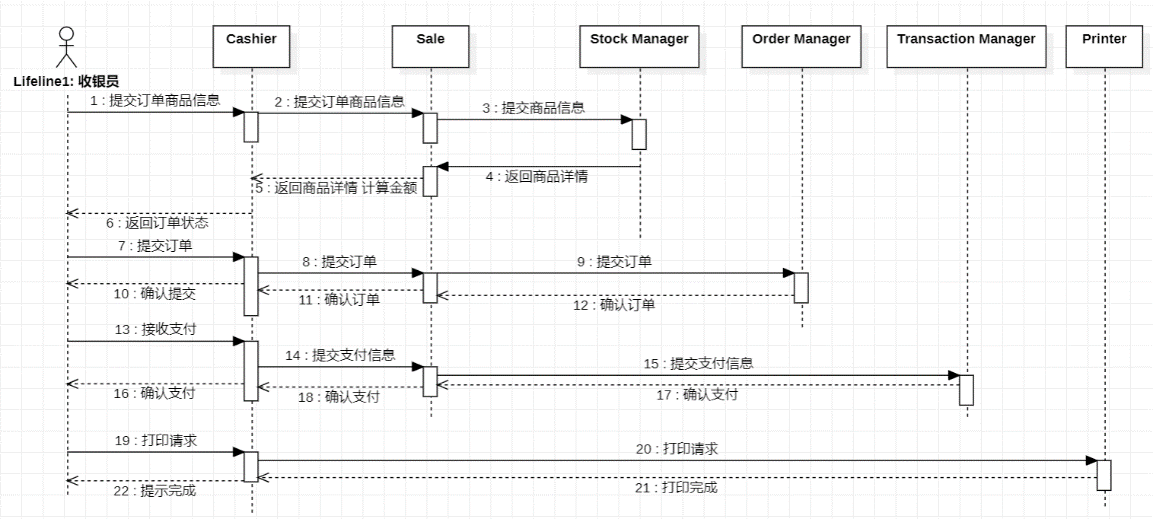


* 时序图

登录：



收银：



* 数据库驱动细节

所有语句均为Prepared Statement。

工具函数：

public ResultSet selectIDQuery(int ID) {  
 try {  
 selectById.setInt(1, ID);  
 // 执行查询  
 selectById.execute();  
 // 获取结果集  
 if(!autoCommit)  
 conn.commit();  
 return selectById.getResultSet();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return null;  
}  
  
public ResultSet selectStringQuery(String name) {  
 try {  
 selectByName.setString(1, name);  
 // 执行查询  
 selectByName.execute();  
 // 获取结果集  
 if(!autoCommit)  
 conn.commit();  
 return selectByName.getResultSet();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return null;  
}  
  
public ResultSet selectStringAlikeQuery(String name) {  
 try {  
 selectAlike.setString(1, "%" + name + "%");  
 // 执行查询  
 selectAlike.execute();  
 // 获取结果集  
 if(!autoCommit)  
 conn.commit();  
 return selectAlike.getResultSet();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return null;  
}  
  
public ResultSet selectRangeQuery(int limit, int offset) {  
 try {  
 selectRange.setInt(1, limit);  
 selectRange.setInt(2, offset);  
 // 执行查询  
 selectRange.execute();  
 // 获取结果集  
 if(!autoCommit)  
 conn.commit();  
 return selectRange.getResultSet();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return null;  
}  
  
public int selectCountQuery() throws IllegalArgumentException {  
 try {  
 // 执行查询  
 count.execute();  
 if(!autoCommit)  
 conn.commit();  
 // 获取结果集  
 ResultSet rs = count.getResultSet();  
 if (!rs.next())  
 throw new IllegalArgumentException("Table corrupted.");  
 return rs.getInt(1);  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 return 0;  
}  
  
public void close() {  
 try {  
 conn.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
  
}  
  
public String getTableName() {  
 return tableName;  
}  
  
public void setTableName(String tableName) throws IllegalArgumentException {  
 if (tableName != null && !tableName.equals("") && !tableName.equals(this.tableName)) {  
 selectRange = createPreparedStatement("SELECT \* FROM pos." + tableName + " LIMIT ? OFFSET ?");  
 count = createPreparedStatement("SELECT COUNT(\*) FROM pos." + tableName);  
 } else {  
 throw new IllegalArgumentException("Table name is null or empty or same as before.");  
 }  
 this.tableName = tableName;  
}  
  
public String getIDName() {  
 return IDName;  
}  
  
public void setIDName(String IDName) throws IllegalArgumentException {  
 if (tableName != null && IDName != null && !tableName.equals("") && !IDName.equals("") && !IDName.equals(this.IDName))  
 selectById = createPreparedStatement("SELECT \* FROM pos." + tableName + " WHERE " + IDName + " = ?");  
 else  
 throw new IllegalArgumentException("Table name or ID name is null or empty or same as before.");  
 this.IDName = IDName;  
}  
  
public String getLabelName() {  
 return labelName;  
}  
  
public void setLabelName(String labelName) {  
 if (tableName != null && labelName != null && !tableName.equals("") && !labelName.equals("") && !labelName.equals(this.labelName)) {  
 selectByName = createPreparedStatement("SELECT \* FROM pos." + tableName + " WHERE " + labelName + " = ?");  
 selectAlike = createPreparedStatement("SELECT \* FROM pos." + tableName + " WHERE " + labelName + " LIKE ?");  
 } else  
 throw new IllegalArgumentException("Table name or label name is null or empty or same as before.");  
 this.labelName = labelName;  
}

由于主要工作在JDBC操作层面上进行，因此事务处理也移动到这里（而非SQL语句中）。通过设置Connection的autoCommit属性，可以取消Statement的自动提交。在这里，通过Java程序代码接管了事务处理过程，可以方便地引入更多控制因素参与到事务过程，“手动”控制事务的开始、提交和回滚。

为避免赘述，这里不妨以TransactionDbEndImpl的其中一个环节为例：

初始化预编译语句：

insert = conn.createPreparedStatement("INSERT INTO pos.transactionlog (TransactionDate, CustomerID, SalesPersonID, TransactionType, PaymentID) VALUES (?, ?, ?, ?, ?)");

观察addTransactionLog：

@Override  
public void addTransactionLog(TransactionLog transactionLog) {  
 try {  
 paymentManager.addPayment(user, transactionLog.getPayment());  
 insert.setDate(1, new java.sql.Date(transactionLog.getDate().getTime()));  
 insert.setInt(2, transactionLog.getCustomer().getID());  
 insert.setInt(3, transactionLog.getSalesPerson().getID());  
 insert.setString(4, transactionLog.getType().toString());  
 insert.setInt(5, transactionLog.getPayment().getID());  
 insert.executeUpdate();  
 conn.getConnection().commit();  
 } catch (Exception e) {  
 e.printStackTrace();  
 try {  
 conn.getConnection().rollback();  
 } catch (SQLException ex) {  
 ex.printStackTrace();  
 }  
 }  
}

在添加TransactionLog时，它会先尝试提交Payment记录：

paymentManager.addPayment(user, transactionLog.getPayment());

在addPayment（PaymentDbEnd）中：

@Override  
public void addPayment(Payment payment) throws IllegalArgumentException{  
 try {  
 insert.setInt(1, payment.getOrderID());  
 insert.setDate(2, new java.sql.Date(payment.getDate().getTime()));  
 insert.setString(3, payment.getMethod().toString());  
 insert.setBigDecimal(4, payment.getChangeAmount());  
 insert.setBigDecimal(5, payment.getAmount());  
 insert.executeUpdate();  
 conn.getConnection().commit();  
 ResultSet rs = lastID.executeQuery();  
 if (rs.next()) {  
 int paymentID = rs.getInt("PaymentID");  
 payment.setID(paymentID);  
 poolFlush.flush(paymentID, payment);  
 }  
 } catch (SQLException e) {  
 e.printStackTrace();  
 try {  
 conn.getConnection().rollback();  
 } catch (SQLException ex) {  
 ex.printStackTrace();  
 }  
 throw new IllegalArgumentException("Payment add failed");  
 }  
}

如果发生了插入错误（例如订单还未提交就进行支付），通过Java的异常捕获机制，将实现回滚，并继续向上级传递异常。

此时，在TransactionDbEndImpl中，程序将捕获到异常，并继续回滚操作。

这样设计还有一个优点。可以注意到，所有DbEndImpl类在处理涉及上下文对其它表的操作行为时，都通过单例实体Manager类调用接口实现，而非直接在自己的connection中执行statement。要理解这一设计，还有一点不能忽略，即每个DbEndImpl都单独维护一个数据库连接对象connection。因为在实际情形中，POS系统处理的不同信息可能分布存储于不同的数据库中（例如交易信息和商品信息可能分属于不同数据库），因此为了在这种情况下在事实上保证“事务”的原子性、隔离性，我采取了这种思路。

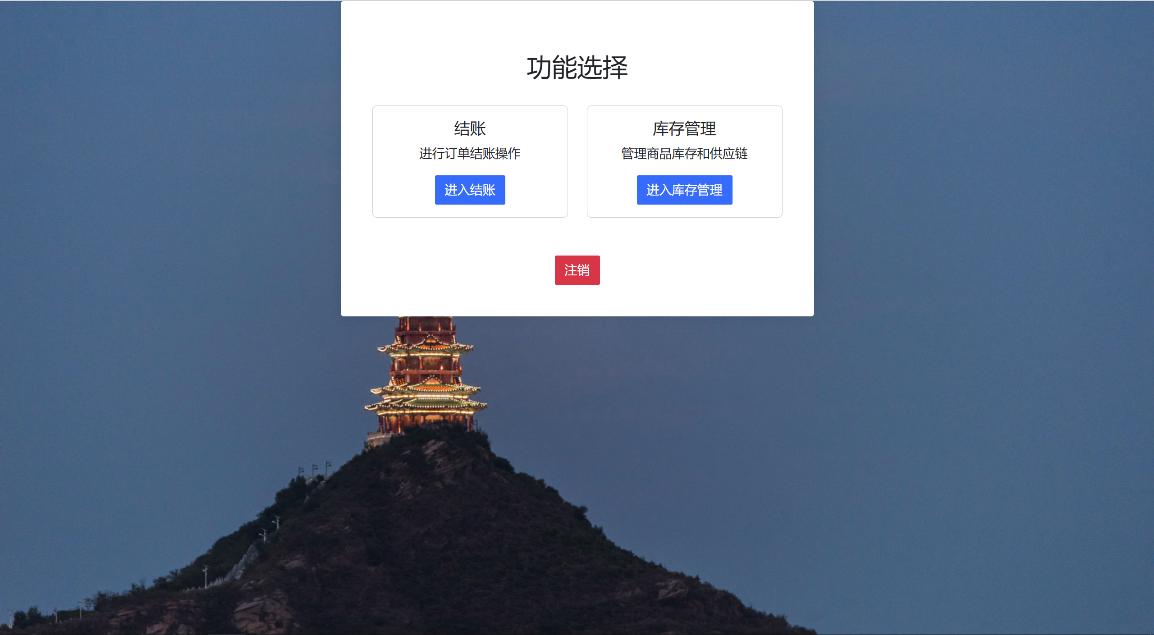
* 前端实现

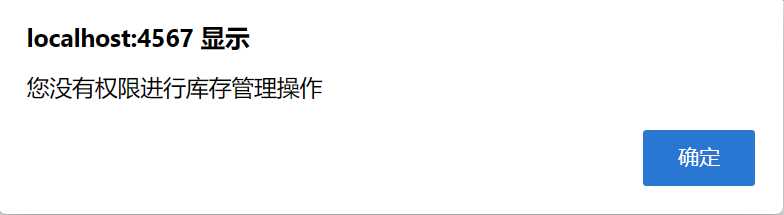
（由于个人精力和能力问题，未覆盖业务模型的全部功能）

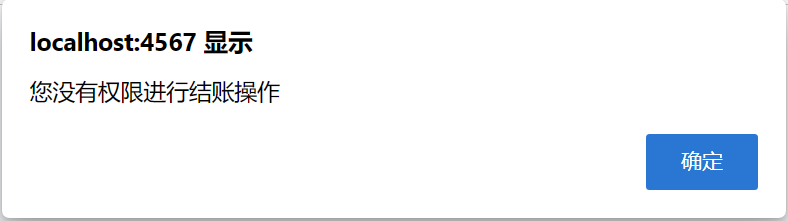
前端页面分为如下几个：

（背景图片从每日必应获取）

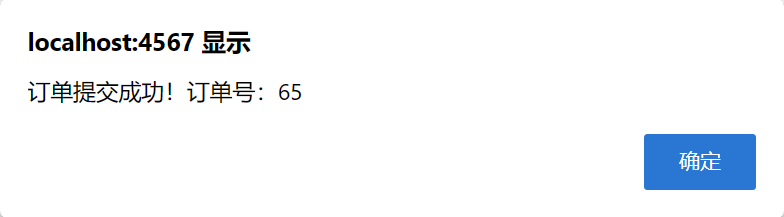
登录页login.html：终端用户输入账户名和密码进行登录。

功能选择页index.html：选择对应功能模块（账户权限约束）或注销。

使用非库存管理或系统管理员账户登录后选择库存管理：

使用非销售员或系统管理员账户登录后选择结账：

结账页pos.html：销售员录入工号，开始销售活动。

一次完整销售过程示例：

查看数据库记录：

TransactionLog：

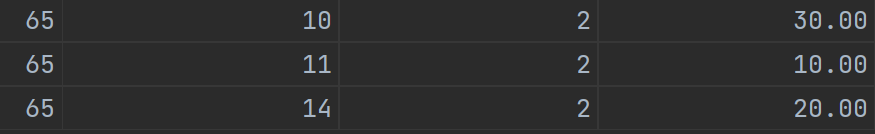


Payments：

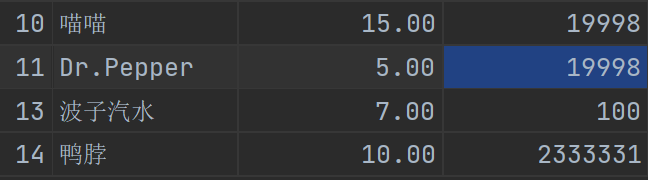


Orders：



OrderDetails：

Products：（库存发生相应变化）



库存管理页stock.html：

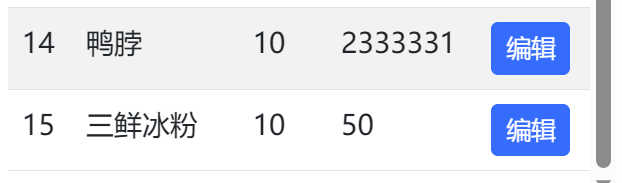


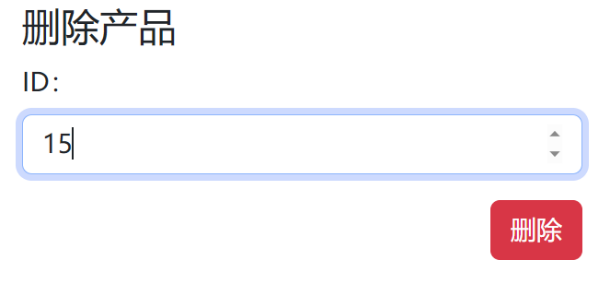
查询用例：

编辑用例：



添加用例：



删除用例：

一些前端代码细节：

广泛采用Ajax异步post操作：

// 更新总金额显示

        function updateTotalAmount() {

            // 发送异步请求到后端获取最新的总金额

            $.post("/pos/get-total-amount")

                .done(function (response) {

                    // 更新总金额的显示

                    $("#totalAmount").val(response);

                })

                .fail(function () {

                    // 处理请求失败的情况

                    alert("请求失败");

                });

        }

实践了两种后端更新前端内容的方法：

1. 后端直接生成html文本，更新到前端：

// 更新商品数量

        function updateQuantity(productId, newQuantity) {

            // 发送异步请求到后端更新商品数量的逻辑

            $.post("/pos/update-quantity", {

                    productId: productId,

                    quantity: newQuantity

                })

                .done(function (response) {

                    // 请求成功后执行相应的操作

                    var tableBody = document.getElementById("checkoutProductList");

                    tableBody.innerHTML = response;

                    // 更新总金额的显示

                    updateTotalAmount();

                })

                .fail(function () {

                    // 处理请求失败的情况

                    alert("请求失败");

                });

        }

private static String updateOrderlist(Request request) {  
 // 获取当前订单的详细内容  
 Cashier cashier = request.session().attribute("cashier");  
 Order order = cashier.getCurrentOrder();  
  
 // 生成订单中每个商品的HTML行  
 StringBuilder productListHtml = new StringBuilder();  
 List<OrderDetail> orderItems = order.getDetail();  
 for (OrderDetail orderItem : orderItems) {  
 Product product = orderItem.getProduct();  
 String productId = String.*valueOf*(product.getProductID());  
 String productName = product.getName();  
 BigDecimal price = product.getPrice();  
 int quantity = orderItem.getQuantity();  
 BigDecimal totalPrice = orderItem.getUnitPrice();  
 // 生成HTML行并追加到商品列表HTML中  
 productListHtml.append("<tr>");  
 productListHtml.append("<td>").append(productId).append("</td>");  
 productListHtml.append("<td>").append(productName).append("</td>");  
 productListHtml.append("<td>").append(price).append("</td>");  
 productListHtml.append("<td>");  
 productListHtml.append("<div class=\"input-group\">");  
 productListHtml.append("<button class=\"btn btn-outline-primary\" onclick=\"decreaseQuantity(").append(productId).append(")\">-</button>");  
 productListHtml.append("<input type=\"number\" class=\"form-control\" id=\"quantity\_").append(productId).append("\" value=\"").append(quantity).append("\">");  
 productListHtml.append("<button class=\"btn btn-outline-primary\" onclick=\"increaseQuantity(").append(productId).append(")\">+</button>");  
 productListHtml.append("</div>");  
 productListHtml.append("</td>");  
 productListHtml.append("<td>").append(totalPrice).append("</td>");  
 productListHtml.append("<td>");  
 productListHtml.append("<button class=\"btn btn-danger\" onclick=\"removeProduct(").append(productId).append(")\">删除</button>");  
 productListHtml.append("</td>");  
 productListHtml.append("</tr>");  
 }  
 return productListHtml.toString();  
}

1. 后端发送描述具体内容的json信息，前端解析生成html元素：

// 获取产品列表

    function getProductList() {

        var fromIndex = parseInt($("#fromIndex").val());

        var toIndex = parseInt($("#toIndex").val());

        // 发起请求获取产品列表数据

        $.post("/stock/get-products", {

                fromIndex: fromIndex,

                toIndex: toIndex

            })

            .done(function (response) {

                // 将响应数据解析为JSON对象

                var productList = JSON.parse(response);

                // 清空产品列表

                $("#productList").empty();

                // 遍历产品数据并添加到列表中

                $.each(productList, function (index, product) {

                    var row = $("<tr>");

                    row.append($("<td>").text(product.id));

                    row.append($("<td>").text(product.name));

                    row.append($("<td>").text(product.price));

                    row.append($("<td>").text(product.stock));

                    row.append($("<td>").html(

                        '<button class="btn btn-sm btn-primary" onclick="editProduct(' + product

                        .id + ', \'' + product.name + '\', ' + product.price + ', ' + product

                        .stock + ')">编辑</button>'

                    ));

                    $("#productList").append(row);

                });

            })

            .fail(function () {

                alert("获取产品列表失败，请重试！");

            });

private Object updateProduct(Request request, Response response) {  
 int productId = Integer.*parseInt*(request.queryParams("id"));  
 String productName = request.queryParams("name");  
 int productQuantity = Integer.*parseInt*(request.queryParams("stock"));  
 BigDecimal productPrice = new BigDecimal(request.queryParams("price"));  
 StockManager stockManager = StockManagerImpl.*Instance*();  
 User user = request.session().attribute("user");  
 try {  
 stockManager.updateProduct(user, new Product(productId, productName, productPrice, productQuantity));  
 } catch (Exception e) {  
 // 400 Bad Request  
 response.status(400);  
 return renderJson(e.getMessage());  
 }  
 return renderJson("success");  
}

总结

本POS系统项目是一个具有一定适用性的数据库管理系统，它可以应用于各种基本的零售和销售场景。通过该项目的实施，我获得了宝贵的经验，深入了解了数据库设计和实施的过程。

项目成功地定义了各种实体（如顾客、订单、产品、销售人员等）和它们之间的关系，使用了合适的数据类型和约束来确保数据的完整性和一致性，并设置了适当的主键和外键以维护数据的完整性和关联性。

在设计过程中，我们对系统的功能和需求进行了详细的分析，包括销售记录与结算、库存管理、财务管理和数据报告等。我们将这些功能与相应的实体和关系进行了结合，以实现系统的完整功能。范式分析确保了数据的高度规范化，减少冗余，使数据的存储和检索过程更加高效和准确。

最终，本项目成功实现了一个稳定、可靠的POS系统，它能够处理销售、库存、数据统计记录等各个方面的业务需求。该系统为用户提供了方便快捷的结账过程，帮助管理者实时掌握库存情况并进行财务分析和决策。总的来说，这个通用的POS系统项目是一个成功的实践，提供了宝贵的经验和技术知识。