

软件设计开发文档

软件名称：铺得清进销存管理系统

版本号：1.0.0

撰写人：田立

撰写日期：2025年11月12日

修订历史

版本号	修订日期	修订人	修订说明
1.0.0	2025-11-12	田立	创建初始文档，完成引言、总体设计和数据库设计部分。

目录

1. 引言 1.1 编写目的 1.2 项目背景 1.3 软件目标 1.4 范围 1.5 参考文献 1.6 术语与缩写
2. 总体设计 2.1 系统概述 2.2 体系结构设计 2.3 功能模块划分 2.4 技术栈选型 2.5 运行环境
3. 数据库设计 3.1 设计概述 3.2 E-R 图 3.3 数据表结构
4. 详细设计与实现 4.1 核心功能：产品管理 4.2 核心功能：销售管理 4.3 核心功能：库存管理 4.4 核心功能：采购管理 4.5 核心功能：数据分析 4.6 通用组件设计
5. 系统部署与维护 5.1 部署图 5.2 部署步骤 5.3 数据备份与恢复
6. 测试 6.1 测试策略 6.2 单元测试 6.3 集成测试
7. 结论

1. 引言

1.1 编写目的

本文档旨在详细阐述“铺得清进销存管理系统”的设计与实现过程。撰写本文档的主要目的如下：

- 软件著作权登记：**作为申请软件著作权所需的核心技术文档，全面、清晰地展示软件的内部结构、功能设计和实现细节，满足鉴别材料的要求。
- 开发指导：**为开发团队提供一个统一、明确的设计蓝图，确保各模块开发工作协调一致，降低沟通成本，提高开发效率。
- 技术存档：**作为项目的重要技术资产，记录软件的设计思想、架构演进和关键技术决策，便于后续的系统维护、功能迭代和技术交接。
- 项目验收：**为项目管理者、测试人员和最终用户提供一个评估软件功能和性能的基准。

通过本文档，相关人员可以深入理解本软件的架构、核心算法、数据结构以及业务逻辑，为软件的整个生命周期管理提供支持。

1.2 项目背景

随着零售和批发业务的快速发展，中小型企业面临着日益复杂的库存管理挑战。传统的手工记账或简单的电子表格管理方式，常常导致以下问题：

- **库存信息不准确**：无法实时更新库存数量，导致超卖或缺货，影响客户满意度和销售机会。
- **运营效率低下**：商品入库、出库、盘点等流程繁琐，耗费大量人力和时间。
- **决策缺乏数据支持**：难以准确统计商品销售情况、分析客户购买行为和预测未来销售趋势，导致采购决策盲目。
- **多渠道管理困难**：线上、线下等多个销售渠道的数据不统一，难以进行一体化管理。

为了解决上述痛点，开发一款现代化、智能化的库存销售管理系统变得至关重要。本项目“Stocko”应运而生，旨在通过数字化手段，为企业提供一个集成化、自动化、智能化的解决方案，以提升运营效率，优化库存成本，并为经营决策提供精准的数据支持。

1.3 软件目标

本软件的主要目标是开发一个稳定、高效、易用的库存销售管理平台，具体包括：

1. **产品全生命周期管理**：实现从商品信息录入、分类、规格定义到库存变动的全程跟踪。
2. **高效的销售流程**：提供快速创建销售订单、管理客户信息、记录销售流水的功能，简化销售操作。
3. **精准的库存控制**：支持多仓库/店铺的库存管理，实现实时库存更新、库存预警、保质期管理和库存盘点。
4. **智能的采购建议**：基于销售数据和库存水平，自动生成采购建议，避免缺货和库存积压。
5. **多维度数据分析**：提供销售额、利润、畅销商品排行、客户贡献度等多维度的数据报表和可视化图表，辅助管理者进行科学决策。
6. **跨平台支持**：基于 Flutter 框架开发，确保软件能够平滑运行在 Android、iOS、Web、Windows、macOS 和 Linux 等多个平台上，满足用户在不同设备上的使用需求。
7. **良好的用户体验**：设计简洁直观的用户界面和流畅的操作流程，降低用户的学习成本。

1.4 范围

本软件的功能范围主要涵盖以下核心模块：

- **产品管理**：包括产品信息、分类、单位、条形码等管理。
- **销售管理**：包括创建销售单、客户管理、销售记录查询等。
- **库存管理**：包括库存查询、入库、出库、盘点、库存异动记录等。
- **采购管理**：包括供应商管理、采购订单创建与跟踪等。
- **数据分析**：提供各类销售和库存相关的统计报表。
- **系统设置**：包括店铺信息、用户权限、数据备份与恢复等。

不在本次开发范围内的功能包括：

- 完整的财务会计功能（如总账、应收应付等）。
- 客户关系管理（CRM）的高级功能（如营销活动、客户关怀等）。
- 与第三方电商平台的深度集成。

1.5 参考文献

- 《软件工程导论（第6版）》
- Flutter 官方文档: <https://flutter.dev/docs>
- Dart 语言规范: <https://dart.dev/guides/language/spec>

- Drift (Moor) 数据库文档: <https://drift.simonbinder.eu/>

1.6 术语与缩写

术语/缩写	中文全称/解释
SKU	Stock Keeping Unit, 库存量单位
UI	User Interface, 用户界面
UX	User Experience, 用户体验
BLoC	Business Logic Component, 一种Flutter状态管理模式
Riverpod	一个用于Flutter/Dart的状态管理库
Drift	一个基于sqlite3的响应式持久化库
API	Application Programming Interface, 应用程序编程接口
CRUD	Create, Read, Update, Delete, 增删改查

2. 总体设计

2.1 系统概述

“铺得清进销存管理系统”是一个基于 Flutter 框架和 Dart 语言开发的跨平台应用程序。系统采用客户端-数据库的架构模式，其中客户端负责用户交互、业务逻辑处理和数据展示，而本地数据库（基于 SQLite）负责所有业务数据的持久化存储。

这种设计使得应用可以在没有网络连接的情况下独立运行，保证了数据的本地性和操作的即时响应。同时，系统也预留了与远程服务器同步的接口，为未来的云服务和多设备数据同步功能奠定了基础。

系统整体遵循功能模块化、代码分层化的设计原则，以实现高内聚、低耦合的目标，从而提高代码的可维护性、可扩展性和可测试性。

2.2 体系结构设计

本系统采用经典的三层架构（Three-Layer Architecture），并结合了 Flutter 社区流行的状态管理模式（如 Riverpod），具体分层如下：

1. 表现层 (Presentation Layer):

- **职责**: 负责直接与用户进行交互，展示数据和接收用户输入。它由一系列的界面组件（Widgets）构成。
- **组成**: 包括各个功能模块的屏幕（Screens）、对话框（Dialogs）、卡片（Cards）等UI元素。
- **技术**: 完全使用 Flutter Widget 构建，不包含任何业务逻辑，其状态由应用逻辑层驱动。

2. 应用/业务逻辑层 (Application/Business Logic Layer):

- **职责**: 作为表现层和数据访问层的桥梁，处理核心业务逻辑、状态管理和用户操作的响应。
- **组成**:

- **Notifier/Provider (基于 Riverpod)**: 管理UI状态，监听用户事件和数据变化，并通知UI进行更新。例如，`CategoryNotifier` 负责管理商品分类的状态。
- **Service**: 封装具体的业务流程和复杂的计算。例如，`SaleService` 负责处理创建销售订单的完整流程。
- **技术**: 使用 Riverpod 进行状态管理，实现依赖注入和业务逻辑与UI的分离。

3. 数据访问层 (Data Access Layer):

- **职责**: 负责数据的持久化和访问，与本地数据库进行交互。它将底层数据源 (SQLite) 的实现细节与上层业务逻辑完全隔离。
- **组成**:
 - **Repository**: 定义了数据操作的接口 (`IProductRepository`)，并提供具体的实现 (`ProductRepository`)。它是业务逻辑层访问数据的唯一入口。
 - **DAO (Data Access Object)**: 数据访问对象，直接执行对数据库表的CRUD操作。例如，`SalesTransactionDao` 负责操作销售流水表。
- **技术**: 使用 Drift 库来定义数据表、执行数据库查询和事务操作。Drift 能够将 Dart 类映射到数据库表，并生成类型安全的代码。

架构图:



2.3 功能模块划分

根据系统目标和范围，我们将软件划分为以下几个核心功能模块：

- **产品模块 (Product)**:
 - **描述**: 管理所有与商品相关的信息。
 - **子功能**: 商品信息 (名称、图片、描述) 管理、商品分类管理、多单位 (如个、箱、斤) 管理、条形码管理、商品导入导出。
- **销售模块 (Sale)**:
 - **描述**: 处理日常的销售业务。
 - **子功能**: 购物车、创建销售订单、客户信息管理、历史销售记录查询、销售退货处理。
- **库存模块 (Inventory)**:

- **描述:** 核心模块，负责跟踪库存的实时变化。
- **子功能:** 实时库存查询、商品入库（采购入库、其他入库）、商品出库（销售出库、报损出库）、库存盘点、库存调拨、保质期预警。
- **采购模块 (Purchase):**
 - **描述:** 管理向供应商采购商品的过程。
 - **子功能:** 供应商信息管理、创建采购订单、跟踪采购订单状态（待入库、已完成）。
- **分析模块 (Analytics):**
 - **描述:** 为经营决策提供数据支持。
 - **子功能:** 销售额统计（按日/周/月）、利润分析、畅销商品排行、客户购买力分析、库存周转率分析。
- **设置模块 (Settings):**
 - **描述:** 提供系统级的配置和管理功能。
 - **子功能:** 店铺/仓库信息管理、用户与权限管理、数据备份与恢复、隐私政策与关于。

2.4 技术栈选型

类别	技术/框架	版本/说明
开发语言	Dart	3.x
UI 框架	Flutter	3.x
状态管理	Riverpod	2.x, 用于依赖注入和状态管理
本地数据库	SQLite	-
数据库ORM	Drift (原Moor)	2.x, 提供类型安全的响应式数据库访问
路由管理	go_router	-
HTTP客户端	Dio	用于未来可能的API交互
测试框架	flutter_test, test	用于单元测试和集成测试
构建系统	Gradle (Android), Xcode (iOS)	-

2.5 运行环境

- **操作系统:** Android 5.0+, iOS 11.0+, Windows 10+, macOS 10.15+, Linux (主流发行版)
- **硬件要求:**
 - 移动端：至少 2GB RAM
 - 桌面端：至少 4GB RAM, 1280x720 分辨率
- **依赖软件:** 无特殊外部依赖，应用打包后可独立运行。

3. 数据库设计

3.1 设计概述

数据库是本系统的核心基础，负责存储所有业务数据。我们选用 SQLite 作为底层数据库引擎，因为它轻量、无需配置、跨平台且性能优异，非常适合单机版桌面和移动应用。

通过使用 Drift 库，我们以声明式的方式在 Dart 代码中定义数据表、关系和查询，Drift 会自动生成类型安全的代码，极大地提高了数据库操作的安全性和开发效率。

数据库设计遵循第三范式（3NF），以减少数据冗余，保证数据的一致性。

3.2 E-R 图

(由于纯文本难以绘制复杂的E-R图，此处使用文字描述核心实体及其关系)

核心实体：

- **Product (产品)**：存储商品基本信息。
- **Category (分类)**：产品的分类。
- **Unit (单位)**：商品的计量单位（如个、箱）。
- **ProductUnit (产品单位)**：产品和单位的关联表，定义了换算关系和不同单位的售价。
- **Customer (客户)**：销售业务中的客户信息。
- **Supplier (供应商)**：采购业务中的供应商信息。
- **SalesTransaction (销售流水)**：记录每一笔销售的总体信息。
- **SalesTransactionItem (销售流水项)**：记录一笔销售中具体每个商品的销售情况。
- **Batch (批次)**：用于跟踪具有保质期或不同成本的同种商品。
- **Inventory (库存)**：记录每个商品（或批次）在特定店铺/仓库的实时库存量。

主要关系：

- 一个 **Category** 可以包含多个 **Product**（一对多）。
- 一个 **Product** 可以有多个 **ProductUnit**（一对多）。
- 一个 **Unit** 可以被多个 **ProductUnit** 使用（一对多）。
- 一个 **Customer** 可以有多笔 **SalesTransaction**（一对多）。
- 一笔 **SalesTransaction** 包含多个 **SalesTransactionItem**（一对多）。
- 一个 **SalesTransactionItem** 关联一个 **ProductUnit**（多对一）。
- 一个 **Product** 可以有多个 **Batch**（一对多）。
- 一个 **Product**（或 **Batch**）在一个店铺有一个 **Inventory** 记录（一对一）。

3.3 数据表结构

以下是部分核心数据表的结构定义（基于Drift的Dart类定义风格）。

1. 产品表 (products)

```
// lib/features/product/domain/model/product.dart
class Products extends Table {
    IntColumn get id => integer().autoIncrement();
    TextColumn get name => text().withLength(min: 1, max: 100)();
    TextColumn get description => text().nullable();
    IntColumn get categoryId => integer().nullable().references(Categories, #id)();
    TextColumn get imageUrl => text().nullable();
    BoolColumn get isActive => boolean().withDefault(const Constant(true))();
    DateTimeColumn get createdAt => dateTime().withDefault(currentDateAndTime)();
}
```

- **说明：**存储商品的核心信息，如名称、描述，并关联到分类表。

2. 产品单位表 (`product_units`)

```
// lib/features/product/domain/model/product_unit.dart
class ProductUnits extends Table {
    IntColumn get id => integer().autoIncrement();
    IntColumn get productId => integer().references(Products, #id)();
    IntColumn get unitId => integer().references(Units, #id)();
    // 基础单位换算率 (例如: 1箱 = 12个, 则'箱'的此值为12)
    RealColumn get conversionRate => real().withDefault(const Constant(1.0))();
    // 是否为基础单位 (例如: '个')
    BoolColumn get isBaseUnit => boolean().withDefault(const Constant(false))();
    // 零售价
    RealColumn get sellingPrice => real().nullable();
    // 采购价
    RealColumn get purchasePrice => real().nullable();
}
```

- **说明：**定义了产品的多种销售/库存单位及其换算关系，是实现多单位功能的核心。

3. 销售流水表 (`sales_transactions`)

```
// lib/features/sale/domain/model/sales_transaction.dart
class SalesTransactions extends Table {
    IntColumn get id => integer().autoIncrement();
    IntColumn get customerId => integer().nullable().references(Customers, #id)();
    DateTimeColumn get transactionDate => dateTime().withDefault(currentDateAndTime)();
    RealColumn get totalAmount => real();
    RealColumn get discount => real().withDefault(const Constant(0.0))();
    RealColumn get finalAmount => real();
    TextColumn get paymentMethod => text().withLength(max: 50)();
    TextColumn get remarks => text().nullable();
}
```

- **说明：**记录每一笔成功交易的宏观信息。

4. 销售流水项表 (`sales_transaction_items`)

```
// lib/features/sale/domain/model/sales_transaction_item.dart
class SalesTransactionItems extends Table {
    IntColumn get id => integer().autoIncrement();
    IntColumn get transactionId => integer().references(SalesTransactions, #id)();
    IntColumn get productUnitId => integer().references(ProductUnits, #id)();
    RealColumn get quantity => real();
    RealColumn get unitPrice => real(); // 成交单价
    RealColumn get subtotal => real();
}
```

- **说明：**记录一笔交易中售出的每一种商品的具体信息。

5. 库存表 (inventories)

```
// lib/features/inventory/domain/model/inventory.dart
class Inventories extends Table {
    IntColumn get id => integer().autoIncrement();
    IntColumn get productId => integer().references(Products, #id)();
    IntColumn get batchId => integer().nullable().references(Batches, #id)(); // 关联批次
    IntColumn get shopId => integer().references(Shops, #id)(); // 关联店铺/仓库
    RealColumn get quantity => real(); // 以基础单位计量的数量
    DateTimeColumn get lastUpdated => dateTime().withDefault(currentDateAndTime)();

    @override
    List<String> get customConstraints => [
        'UNIQUE (product_id, batch_id, shop_id)'
    ];
}
```

- **说明：**系统的核心表之一，实时记录每个商品在每个店铺的库存量。`customConstraints`确保了同一商品（或批次）在同一店铺只有一条库存记录。

4. 详细设计与实现

本章将深入探讨系统中各个核心功能模块的详细设计和代码实现。我们将结合具体的代码片段，阐述业务逻辑、数据流转以及与用户界面的交互过程。

4.1 核心功能：产品管理

产品管理是整个系统的基础，它负责定义和维护可供销售和管理库存的商品信息。

4.1.1 功能描述

- **创建、编辑和查看商品：**用户可以录入商品的详细信息，包括名称、所属分类、描述和图片。
- **多单位管理：**系统支持为同一商品设置多个单位（如“瓶”和“箱”），并定义它们之间的换算率。每个单位都可以有独立的售价和采购价。这是本软件的一个关键特性。
- **分类管理：**用户可以创建商品分类，便于对商品进行组织和筛选。
- **条码关联：**可以为不同的产品单位关联一个或多个条形码，支持扫码枪快速识别。

4.1.2 数据流与状态管理

产品管理模块的数据流严格遵循2.2节中定义的分层架构。

1. **用户操作：**用户在UI界面（如 `ProductEditScreen`）上输入信息并点击“保存”。
2. **UI层 -> 逻辑层：**UI Widget 调用应用逻辑层中的 `Notifier` 或 `Service` 相应的方法，例如 `ref.read(productServiceProvider).saveProduct(productData)`。

3. **逻辑层处理**: `ProductService` 接收到数据后，进行数据校验和业务逻辑处理。
4. **逻辑层 -> 数据层**: `ProductService` 调用 `ProductRepository` 的方法（如 `saveProduct`）来持久化数据。
5. **数据层执行**: `ProductRepository` 内部再调用 `ProductDao`，后者通过 Drift 框架将数据写入 SQLite 数据库的 `products` 和 `product_units` 表中。
6. **状态更新**: 数据成功保存后，逻辑层的 `Notifier`（如 `productListProvider`）会重新获取最新的产品列表，并通知 UI 层进行刷新，展示最新的数据。

4.1.3 核心代码实现：多单位管理

多单位管理的核心在于 `product_units` 表以及围绕它展开的业务逻辑。以下是 `ProductUnitRepository` 中更新产品单位信息的核心方法，它展示了如何在一个事务中处理复杂的更新逻辑。

文件路径: `lib/features/product/data/repository/product_unit_repository.dart`

```
// 伪代码，结合项目结构和Drift用法进行阐述
// public Future<void> updateProductUnits(int productId, List<ProductUnit> units)
// 在 ProductRepository 中实现类似逻辑
Future<void> saveProductWithUnits(Product product, List<ProductUnit> units) async {
  return transaction(() async {
    // 步骤 1: 保存或更新产品主信息
    final productId = await productsDao.insertOnConflictUpdate(product);

    // 步骤 2: 获取该产品当前所有的单位ID
    final existingUnits = await productUnitsDao.getUnitsForProduct(productId);
    final existingUnitIds = existingUnits.map((e) => e.id).toSet();

    // 步骤 3: 遍历用户提交的最新单位列表
    final updatedUnitIds = <int>{};
    for (final unit in units) {
      // 为单位关联正确的产品ID
      final unitWithProduct = unit.copyWith(productId: productId);

      // 步骤 3.1: 插入或更新单位信息
      final savedUnit = await
        productUnitsDao.insertOnConflictUpdate(unitWithProduct);
      updatedUnitIds.add(savedUnit.id);
    }

    // 步骤 4: 计算需要被删除的单位
    // (即存在于旧列表但不存在于新列表中的单位)
    final unitsToDelete = existingUnitIds.difference(updatedUnitIds);
    if (unitsToDelete.isNotEmpty) {
      // 步骤 4.1: 执行删除操作
      await productUnitsDao.deleteUnitsByIds(unitsToDelete.toList());
    }
  });
}
```

- **代码解析:**

- 该方法使用 `transaction` 来确保所有数据库操作的原子性：要么全部成功，要么全部回滚。这对于保证数据一致性至关重要。
- 通过 `insertOnConflictUpdate` 实现“有则更新，无则插入”(Upsert)，简化了代码逻辑。
- 通过比较新旧单位列表的ID集合，精确地计算出哪些单位是本次操作中被用户删除的，并执行相应的数据库删除操作。
- 这个过程完整地体现了对一个复杂业务对象（包含子对象的商品）的持久化处理逻辑。

4.2 核心功能：销售管理

销售管理模块是系统产生业务价值的直接体现，它覆盖了从开单到收款的全过程。

4.2.1 功能描述

- **购物车：**提供一个直观的界面，允许用户通过搜索或扫码将商品添加到购物车。
- **创建销售单：**用户可以为购物车中的商品选择客户、输入折扣、选择支付方式，并最终完成结账，生成一笔销售流水。
- **客户选择：**支持从现有客户列表中选择，或快速添加新客户。
- **销售记录：**可以按时间、客户等条件查询历史销售记录，并查看每笔订单的详细信息。

4.2.2 核心界面实现：创建销售屏幕

创建销售的核心UI位于 `CreateSaleScreen`。它通常由几个部分组成：顶部的商品搜索栏、中间的购物车商品列表、底部的客户选择和结算信息区域。

文件路径: lib/features/sale/presentation/screens/create_sale_screen.dart

```
// 伪代码，展示Widget结构和状态交互
class CreateSaleScreen extends ConsumerWidget {
  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // 监听购物车状态
    final cartItems = ref.watch(saleCartProvider);
    // 监听客户选择状态
    final selectedCustomer = ref.watch(selectedCustomerProvider);

    return Scaffold(
      appBar: AppBar(title: Text('创建销售单')),
      body: Column(
        children: [
          // 1. 商品搜索与添加区域
          ProductSearchView(),

          // 2. 购物车商品列表
          Expanded(
            child: ListView.builder(
              itemCount: cartItems.length,
              itemBuilder: (context, index) {
                final item = cartItems[index];
                // 使用 SaleItemCard Widget 展示每个商品
              }
            )
          )
        ],
      )
    );
  }
}
```

```
        return SaleItemCard(
            item: item,
            onQuantityChanged: (newQty) {
                // 调用Notifier更新商品数量
                ref.read(saleCartProvider.notifier).updateQuantity(item.id,
newQty);
            },
            onRemove: () {
                // 调用Notifier移除商品
                ref.read(saleCartProvider.notifier).removeItem(item.id);
            },
        );
    },
),
),

// 3. 结算信息区域
SaleSummarySection(
    customer: selectedCustomer,
    totalAmount: ref.watch(saleCartTotalProvider), // 动态计算总价
    onSelectCustomer: () => context.go('/customer-selection'), // 跳转到客
户选择页
onCheckout: () {
    // 执行结算
    ref.read(saleServiceProvider).createSale(
        items: cartItems,
        customer: selectedCustomer,
        // ...其他结算信息
    );
},
),
],
),
);
}
}
```

- 代码解析：

- 该界面是一个 `ConsumerWidget`，表明它依赖于 Riverpod 进行状态管理。
 - 通过 `ref.watch` 监听 `saleCartProvider` 的变化。当购物车内容（如商品数量、种类）发生改变时，UI会自动重建，实时反映最新状态。
 - `SaleItemCard` 作为一个独立的组件，封装了购物车中单个条目的显示和操作逻辑，体现了UI开发的组件化思想。
 - 点击结算按钮时，调用 `SaleService` 的 `createSale` 方法，将业务逻辑处理委托给应用逻辑层，保持了表现层的职责单一。

4.2.3 核心业务逻辑：创建销售流水

SaleService 中的 `createSale` 方法是处理销售业务的核心，它协调了多个数据表的写入操作。

文件路径: lib/features/sale/application/service/sale_service.dart

```
// 伪代码
class SaleService {
    final SalesTransactionRepository _transactionRepo;
    final InventoryRepository _inventoryRepo;

    SaleService(this._transactionRepo, this._inventoryRepo);

    Future<void> createSale(
        {required List<SaleCartItem> items, Customer? customer, ...}) async {

        return _transactionRepo.transaction(() async {
            // 步骤 1: 创建销售流水主记录 (SalesTransaction)
            final transaction = SalesTransaction(
                customerId: customer?.id,
                totalAmount: calculateTotal(items),
                // ...
            );
            final transactionId = await _transactionRepo.createTransaction(transaction);

            // 步骤 2: 遍历购物车中的每一个商品
            for (final item in items) {
                // 步骤 2.1: 创建销售流水项 (SalesTransactionItem)
                final transactionItem = SalesTransactionItem(
                    transactionId: transactionId,
                    productUnitId: item.productUnit.id,
                    quantity: item.quantity,
                    unitPrice: item.unitPrice,
                    // ...
                );
                await _transactionRepo.createTransactionItem(transactionItem);

                // 步骤 2.2: 更新库存 (核心)
                // 将销售数量换算成基础单位的数量
                final baseUnitQuantity = item.quantity * item.productUnit.conversionRate;
                await _inventoryRepo.decreaseStock(
                    productId: item.productUnit.productId,
                    quantity: baseUnitQuantity,
                );
            }
        });
    }
}
```

• 代码解析:

- 同样地，整个过程被包裹在一个数据库事务中，以确保销售记录的生成和库存的扣减要么同时成功，要么同时失败。
- 职责分离：**`SaleService` 负责编排整个业务流程，但具体的数据库操作分别委托给了 `SalesTransactionRepository` 和 `InventoryRepository`，符合单一职责原则。
- 库存更新：**在记录销售明细后，服务会立即调用 `InventoryRepository` 的 `decreaseStock` 方法来扣减相应商品的库存。这是保证库存数据准确性的关键一步。

- **单位换算**: 在扣减库存时，代码将销售单位的数量（如2箱）乘以换算率，转换成库存管理所用的基础单位（如24瓶），体现了多单位设计的严谨性。

4.3 核心功能：库存管理

库存管理是本系统的基石，它确保了所有库存相关数据的准确性和实时性，是连接销售、采购等其他模块的枢纽。

4.3.1 功能描述

- **实时库存查询**: 用户可以随时查看任一商品在所有店铺或单个店铺的当前库存量。
- **库存变更**: 系统自动处理因销售导致的出库。同时，用户可以手动进行采购入库、盘点校正、库存调拨、报损等操作，所有操作都会生成明确的库存异动流水。
- **保质期管理**: 对于有保质期的商品，系统支持录入生产日期和保质期天数，并提供临期商品预警功能。

4.3.2 核心业务逻辑：库存更新服务

库存的所有变动都由 `InventoryService` 或底层的 `InventoryRepository` 来统一处理，以确保逻辑的集中和数据的一致性。以下是库存变更的核心方法 `updateStock` 的设计思路。

文件路径: `lib/features/inventory/application/inventory_service.dart` (或 `InventoryRepository`)

```
// 伪代码
class InventoryRepository {
    // ... Dao a

    // 统一的库存更新入口
    Future<void> updateStock({
        required int productId,
        required double changeQuantity, // 正数为增加，负数为减少
        required InventoryTransactionType type, // 异动类型：销售、采购、盘点等
        String? remarks,
        int? batchId,
        int? shopId,
    }) async {
        return transaction(() async {
            // 步骤 1: 找到或创建对应的库存记录
            final inventory = await inventoryDao.findOrCreate(productId, batchId,
shopId);

            // 步骤 2: 计算新库存
            final newQuantity = inventory.quantity + changeQuantity;
            if (newQuantity < 0) {
                throw Exception('库存不足，操作失败！');
            }

            // 步骤 3: 更新库存表
            await inventoryDao.updateQuantity(inventory.id, newQuantity);

            // 步骤 4: 记录库存异动流水 (非常重要)
            final transactionRecord = InventoryTransaction(

```

```
        productId: productId,
        batchId: batchId,
        shopId: shopId,
        changeQuantity: changeQuantity,
        newQuantity: newQuantity,
        type: type,
        remarks: remarks,
        transactionDate: DateTime.now(),
    );
    await inventoryTransactionDao.insert(transactionRecord);
});
}

// 封装好的增加库存方法
Future<void> increaseStock({int productId, double quantity, ...}) {
    return updateStock(productId: productId, changeQuantity: quantity, ...);
}

// 封装好的减少库存方法
Future<void> decreaseStock({int productId, double quantity, ...}) {
    return updateStock(productId: productId, changeQuantity: -quantity, ...);
}
}
```

- **代码解析:**

- **统一入口:** 所有引起库存变化的操作最终都会调用 `updateStock` 这个核心方法。这种设计避免了在代码库的多个地方分散地修改库存，极大地降低了出错的风险。
- **库存防超卖:** 在更新库存前，会检查变更后的库存量是否会小于零。这是防止商品超卖的关键控制点。
- **异动流水:** 每次库存变更，系统都会在 `inventory_transactions` 表中插入一条详细的流水记录。这为后续的库存追溯、成本核算和问题排查提供了不可或缺的数据支持。
- **原子性:** 更新库存量和插入流水这两个操作被包裹在同一个数据库事务中，确保了二者的一致性。

4.4 核心功能：采购管理

采购管理模块负责处理与供应商和商品采购相关的业务流程。

4.4.1 功能描述

- **供应商管理:** 维护供应商的基本信息，如名称、联系方式、地址等。
- **采购订单:** 用户可以创建采购订单，指定供应商、采购的商品、数量和采购单价。
- **采购入库:** 当采购的商品到货后，用户可以基于采购订单执行“一键入库”操作，系统会自动增加相应商品的库存。

4.4.2 核心业务逻辑：采购入库

采购入库是连接采购模块和库存模块的关键流程。当用户在 `PurchaseOrderDetailScreen` 点击“入库”按钮时，会触发 `PurchaseService` 中的 `receivePurchaseOrder` 方法。

文件路径: lib/features/purchase/application/service/purchase_service.dart (设想)

```
// 伪代码
class PurchaseService {
    final PurchaseRepository _purchaseRepo;
    final InventoryService _inventoryService;

    PurchaseService(this._purchaseRepo, this._inventoryService);

    Future<void> receivePurchaseOrder(int purchaseOrderId) async {
        // 步骤 1: 获取采购订单及其所有订单项
        final order = await _purchaseRepo.getOrderById(purchaseOrderId);
        final orderItems = await _purchaseRepo.getItemsByOrderId(purchaseOrderId);

        // 步骤 2: 遍历所有采购项, 逐个添加入库
        for (final item in orderItems) {
            // 调用库存服务, 增加库存
            await _inventoryService.increaseStock(
                productId: item.productId,
                quantity: item.quantity, // 假设单位已是基础单位
                type: InventoryTransactionType.purchase, // 异动类型为采购入库
                remarks: '采购订单 #${order.id} 入库',
            );
        }

        // 步骤 3: 更新采购订单状态为“已完成”
        await _purchaseRepo.updateOrderStatus(purchaseOrderId, OrderStatus.completed);
    }
}
```

- **代码解析:**

- **服务协作:** PurchaseService 作为一个协调者, 本身不直接处理库存逻辑, 而是调用 InventoryService 提供的标准服务。这完美体现了分层架构中服务之间协作的模式。
- **流程清晰:** 整个业务流程被清晰地分解为“获取数据”、“处理业务（循环增库存）”、“更新状态”三个步骤, 代码易于理解和维护。

4.5 核心功能：数据分析

数据分析模块旨在将原始的业务数据转化为有价值的商业洞察。

4.5.1 功能描述

- **销售看板:** 以图表形式展示今日/本周/本月的关键指标, 如销售总额、订单数、利润等。
- **排行分析:** 提供商品销售额排行、销量排行, 帮助商家识别畅销和滞销品。
- **趋势分析:** 展示销售额随时间变化的趋势图, 辅助预测和备货。

4.5.2 核心代码实现：销售额排行查询

数据分析功能严重依赖于高效的数据库查询。Drift 库支持复杂的聚合查询和 group by 子句, 非常适合用于构建分析报表。

文件路径: lib/features/analytics/data/repository/sales_analytics_repository.dart

```
// 伪代码
class SalesAnalyticsRepository {
    final AppDatabase _db;
    SalesAnalyticsRepository(this._db);

    // 获取指定时间范围内的商品销售额排行
    Future<List<ProductRankingResult>> getTopSellingProducts({
        required DateTime start,
        required DateTime end,
        int limit = 10,
    }) {
        // 联接销售流水项、销售流水、产品单位和产品表
        final query = _db.select(_db.salesTransactionItems)
            ..join([
                innerJoin(
                    _db.salesTransactions,
                    _db.salesTransactions.id.equalsExp(_db.salesTransactionItems.transactionId),
                ),
                innerJoin(
                    _db.productUnits,
                    _db.productUnits.id.equalsExp(_db.salesTransactionItems.productUnitId),
                ),
                innerJoin(
                    _db.products,
                    _db.products.id.equalsExp(_db.productUnits.productId),
                ),
            ],
        );
    }

    // 应用时间范围过滤
    query.where(_db.salesTransactions.transactionDate.isBetween(start, end));

    // 定义聚合列: 总销售额和总销量
    final totalRevenue = _db.salesTransactionItems.subtotal.sum();
    final totalQuantity = _db.salesTransactionItems.quantity.sum();

    // 按产品ID和名称分组
    query.groupBy([_db.products.id, _db.products.name]);

    // 按总销售额降序排序
    query.orderBy([OrderingTerm(expression: totalRevenue, mode: OrderingMode.desc)]);

    // 限制返回结果数量
    query.limit(limit);

    // 执行查询并映射到结果DTO
    return query.map((row) {
        return ProductRankingResult(
            productId: row.read(_db.products.id)!,
            productName: row.read(_db.products.name)!,
        );
    });
}
```

```
        totalRevenue: row.read(totalRevenue)!,  
        totalQuantity: row.read(totalQuantity)!,  
    );  
}).get();  
}  
}
```

• 代码解析:

- **复杂查询**: 该查询通过 `join` 连接了四张表，以从销售记录追溯到商品信息。
- **聚合与分组**: 使用 `sum()` 函数进行聚合计算，并通过 `groupBy` 对每个商品的数据进行汇总。这是所有报表查询的典型模式。
- **类型安全**: 借助 Drift，整个复杂的SQL查询都是在Dart中以类型安全的方式构建的，编译器可以检查出大多数拼写错误或逻辑错误。
- **可读性**: 相比于拼接原始SQL字符串，这种链式调用的方式使得查询的意图（过滤、分组、排序）更加清晰。

4.6 通用组件设计

为了提高开发效率和UI一致性，系统设计了许多可复用的通用组件。

- **LoadingWidget**: 一个标准的加载指示器，用于在数据加载时向用户显示，避免界面空白。
- **ErrorWidget**: 当发生错误（如网络请求失败、数据加载错误）时，显示统一的错误提示和“重试”按钮。
- **CustomFormField**: 封装了 `TextField`，并集成了统一的样式、标签和验证逻辑。
- **ResponsiveLayout**: 一个响应式布局组件，能够根据屏幕宽度（移动端、平板、桌面端）调整其子组件的布局方式，是实现跨平台UI的核心。

5. 系统部署与维护

本章主要描述应用的部署结构、安装步骤以及后续的维护策略。

5.1 部署图

本系统为单机版应用程序，其部署结构非常简洁。所有组件，包括UI、业务逻辑和数据库，都打包在同一个应用程序安装包内，直接运行在用户的设备上。

部署示意图:





- **说明:** 应用程序与SQLite数据库文件直接在用户设备的文件系统上进行交互。数据存储在本地，不依赖网络连接。

5.2 部署步骤

得益于 Flutter 的跨平台能力，可以将同一套代码库编译成适用于不同操作系统的安装包。

5.2.1 Android 部署

1. **构建APK/AAB:** 在项目根目录下运行以下命令：

```
flutter build apk --release  
# 或者构建用于Google Play商店的App Bundle  
flutter build appbundle --release
```

2. **安装:** 命令执行成功后，会在 `build/app/outputs/flutter-apk/` 目录下生成 `app-release.apk` 文件。将此文件传输到Android设备上，直接点击安装即可。

5.2.2 Windows 部署

1. **构建安装包:** 运行以下命令：

```
flutter build windows --release
```

2. **安装:** 命令执行后，在 `build/windows/runner/Release/` 目录下会生成一个包含 `.exe` 可执行文件和所需 `.dll` 文件的文件夹。可以将整个文件夹打包成压缩包分发，用户解压后直接运行 `.exe` 文件即可启动程序。也可以使用 Inno Setup 等工具制作成标准的Windows安装程序。

5.2.3 其他平台

iOS, macOS, Linux 的部署流程与上述类似，均可通过 `flutter build <platform>` 命令生成对应平台的发布产物。

5.3 数据备份与恢复

数据是用户的核心资产，因此提供可靠的备份与恢复机制至关重要。

- **备份逻辑:**

1. 提供一个“备份数据”的功能按钮。
2. 当用户点击时，程序首先会安全地关闭数据库连接，以确保数据文件处于一致状态。

3. 然后，程序将定位到 SQLite 数据库文件（通常在应用的私有目录下，如 `getApplicationDocumentsDirectory()`）。
4. 将该数据库文件（例如 `app_data.db`）复制到用户选择的一个安全位置（如“下载”文件夹或外部存储设备）。
5. 为了增加安全性，可以考虑在复制前对数据库文件进行加密和压缩。

- **恢复逻辑：**

1. 提供“恢复数据”功能，并明确提示用户这将覆盖当前所有数据。
2. 用户选择之前备份的数据库文件。
3. 程序验证文件的有效性（例如，通过文件头或解密尝试）。
4. 关闭现有数据库连接，用备份文件替换掉应用当前使用的数据库文件。
5. 重新初始化并打开数据库连接，加载恢复后的数据。

6. 测试

为了保证软件的质量、稳定性和可靠性，我们制定了多层次的测试策略。

6.1 测试策略

我们采用金字塔模型作为测试策略的指导思想，自下而上分为单元测试、集成测试和端到端（E2E）/UI测试。

- **单元测试 (Unit Tests)**：数量最多，专注于测试单一函数、方法或类的逻辑是否正确。运行速度快，反馈周期短。
- **集成测试 (Integration Tests)**：测试多个模块协同工作时的正确性，特别是服务层与数据访问层的交互。
- **UI测试 (Widget Tests / E2E Tests)**：模拟用户操作，验证UI显示和交互流程是否符合预期。

6.2 单元测试

单元测试主要针对无外部依赖（特别是无UI和数据库依赖）的纯逻辑代码。

示例：测试一个数据模型中的计算属性

文件路径: `test/features/product/domain/model/product_unit_test.dart`

```
import 'package:test/test.dart';
import 'package:stocko_app/features/product/domain/model/product_unit.dart';

void main() {
  group('ProductUnit', () {
    test('isBaseUnit should be true when conversionRate is 1.0', () {
      // 准备 (Arrange)
      final unit = ProductUnit(id: 1, productId: 1, unitId: 1, conversionRate:
      1.0);

      // 执行 (Act)
      final isBase = unit.isBaseUnit;

      // 断言 (Assert)
    });
  });
}
```

```
        expect(isBase, isTrue);
    });

    test('isBaseUnit should be false when conversionRate is not 1.0', () {
        // 准备 (Arrange)
        final unit = ProductUnit(id: 1, productId: 1, unitId: 1, conversionRate:
12.0);

        // 执行 (Act)
        final isBase = unit.isBaseUnit;

        // 断言 (Assert)
        expect(isBase, isFalse);
    });
});
}
```

- **说明：**这个测试用例验证了 `ProductUnit` 模型中 `isBaseUnit` 这个计算属性的逻辑是否正确，它不依赖任何外部系统，可以快速执行。

6.3 集成测试

集成测试的重点是验证应用逻辑层和数据访问层的交互。在本项目中，主要是测试 `Repository` 是否能正确地与 `Drift/SQLite` 数据库进行交互。

示例：测试 `ProductRepository` 的增删改查

文件路径: `test/features/product/data/repository/product_repository_test.dart`

```
import 'package:flutter_test/flutter_test.dart';
import 'package:stocko_app/core/database/database.dart'; // 假设这是Drift数据库实例
import
'package:stocko_app/features/product/data/repository/product_repository.dart';

void main() {
    late AppDatabase database;
    late ProductRepository repository;

    setUp(() {
        // 使用内存中的数据库进行测试，避免污染真实数据
        database = AppDatabase.inMemory();
        repository = ProductRepository(database.productsDao);
    });

    tearDown(() async {
        await database.close();
    });

    test('create and get product', () async {
        // 准备 (Arrange)
        final newProduct = Product(name: '测试商品', categoryId: 1);
        final createdProduct = await repository.create(newProduct);
        expect(createdProduct.id, isNot(1));
    });
}
```

```
// 执行 (Act)
final productId = await repository.createProduct(newProduct);
final fetchedProduct = await repository.getProductById(productId);

// 断言 (Assert)
expect(fetchedProduct, isNotNull);
expect(fetchedProduct!.name, '测试商品');
});

}
```

- **说明:**

- `setUp` 和 `tearDown` 函数确保每个测试用例都在一个干净、独立的环境中运行。
- 通过 `AppDatabase.inMemory()` 创建一个内存数据库，使得测试可以快速执行，并且不会在文件系统上留下垃圾数据。
- 测试流程模拟了“创建一个产品，然后通过ID取回它，最后验证取回的产品信息是否正确”的完整闭环，验证了 `create` 和 `get` 两个方法的协同工作。

7. 结论

本文档详细阐述了“铺得清进销存管理系统”的设计理念、系统架构、功能实现、部署策略和测试方法。该系统基于 Flutter 框架，采用分层架构和模块化设计，实现了产品、销售、库存、采购等一系列核心功能，并特别针对多单位管理、实时库存同步等业务难点提供了稳健的解决方案。

通过本文档的撰写，我们系统地梳理了软件的全部技术细节，展示了其主要代码的原创性和复杂性，完全满足软件著作权登记申请对鉴别材料的要求。

展望未来，本系统具备良好的可扩展性。基于当前稳固的单机版架构，可以平滑地演进至“云+端”模式，增加多设备数据同步、团队协作、线上商城对接等高级功能，以适应更广泛的商业需求。