跨境电商仓库管理系统 - 技术设计文档

文档版本

- **版本**: 1.3 (评审修订版)
- **日期**: 2025年12月10日
- **作者**: 技术架构团队
- **状态**: 待开发
- **修订记录**:
    - v1.0: 初始草案
    - v1.1: 调整技术栈为 Python/APIFlask，对齐项目统一技术框架
    - v1.2: 补充跨境电商与供应链视角的设计增强
    - v1.3: 内部专家评审后修订：增加库位管理、汽配特性支持、汇率策略，并将 PDA 功能列为技术预备项。

1. 项目概述

1.1 业务背景

公司作为跨境电商企业，仓库管理具有多维度、多形态的特征。实际业务中，仓库从不同角度可分为：

1. **地理维度**：
    - **国内仓库**：位于国内，通常作为总仓或集货仓。
    - **海外仓库**：位于目标市场国（如美、德、日），用于本地发货。

2. **管理/权属维度**：
    - **自营仓库**：公司直接管理，使用本系统完整WMS功能（库位、PDA作业）。
    - **第三方仓库 (3PL)**：外部服务商管理（如FBA、谷仓、万邑通），本系统通过 API 同步库存与指令。
3. **形态维度**：
    - **实体仓**：具备物理实体的仓库。
    - **虚拟仓**：供应链各环节的**业务缓冲区**，用于：
        - **销售端**：库存分配与渠道隔离（支持超卖规则）。
        - **采购端**：采购计划汇聚与需求缓冲（应对长周期备货）。

1.2 核心需求

1. **多维仓库矩阵管理**：支持“国内/海外”+“自营/三方”的灵活组合。
2. **虚拟仓全链路缓冲**：
    - 销售侧：基于规则的库存分配与超卖控制。
    - 采购侧：作为采购计划的容器，管理各团队的备货需求。
3. 实时同步第三方仓库库存。
4. 提供灵活的库存分配规则配置。
5. 支持多团队协作和权限控制。
6. **(新增)** 支持精细化库位管理与汽配行业特性（通用件、体积重）。

2. 系统架构设计

2.1 整体架构图 (Python Modular Monolith)

我们采用**模块化单体架构**，在保持代码库统一（Monorepo）的前提下，通过 Blueprint 实现业务解耦。

```
Parse error on line 19:
...WebContainer --> DB[(PostgreSQL 主库)]
----------------------^
Expecting 'SPACE', 'GRAPH', 'DIR', 'TAGEND', 'TAGSTART', 'UP', 'DOWN',
'subgraph', 'end', 'MINUS', '--', '==', 'STR', 'STYLE', 'LINKSTYLE',
'CLASSDEF', 'CLASS', 'CLICK', 'DEFAULT', 'NUM', 'PCT', 'COMMA',
'ALPHA', 'COLON', 'BRKT', 'DOT', 'PUNCTUATION', 'UNICODE_TEXT', 'PLUS',
'EQUALS', 'MULT', got 'PS'
```

2.2 技术栈选择

后端技术栈 (符合 is-vue-admin 规范)

- **语言**: Python 3.10+
- **Web框架**: APIFlask (基于 Flask + Marshmallow)
- **ORM**: SQLAlchemy 2.0 (Mapped Class 风格)
- **数据验证**: Marshmallow / Webargs
- **异步任务**: Celery 5.x
- **消息中间件**: Redis (作为 Broker 和 Result Backend)
- **应用服务器**: Gunicorn
- **数据库**: PostgreSQL 15+ (利用分区表和 JSONB 特性)

前端技术栈

- **框架**: Vue 3 + TypeScript + Vite
- **架构**: Vben Admin 5.0 (Monorepo)
- **UI库**: Ant Design Vue 4.x
- **状态管理**: Pinia
- **数据表格**: Vxe-Table (通过 Adapter 适配)

2.3 服务拆分与演进预案

当前采用**模块化单体架构**，但在模块边界上预留未来演进为独立服务的能力：

- **潜在服务边界**：
  - `warehouse-service`：仓库与渠道配置、业务类型与监管类型管理
  - `inventory-service`：库存余额、库存成本、库存单据（入库/出库/调拨/调整）
  - `wms-sync-service`：第三方仓/平台库存同步、差异对账与告警
- **演进策略**：
  - 初期在同一代码库/进程中以 Blueprint 形式解耦，只通过 Service 层交互
  - 当单模块 QPS/复杂度达到阈值时，可将对应 Blueprint 抽离为独立进程，通过 HTTP/RPC 与主系统通信
  - 日志、监控、认证等横切能力在单体阶段统一实现，拆分后通过 Sidecar / 网关透传

3. 数据库设计

3.1 核心表结构映射 (SQLAlchemy 2.0)

(v1.3 更新：新增 `WarehouseLocation` 表，完善 `Stock` 关联，增加汽配相关字段)

```
from app.extensions import db
from sqlalchemy.orm import Mapped, mapped_column
from sqlalchemy.dialects.postgresql import JSONB, ARRAY
from datetime import datetime
from decimal import Decimal


class Warehouse(db.Model):
    __tablename__ = 'warehouses'

    id: Mapped[int] = mapped_column(primary_key=True)
    code: Mapped[str] = mapped_column(db.String(50), unique=True)
    name: Mapped[str] = mapped_column(db.String(100))

    # 核心分类属性 (v1.3 重构)
    # 1. 仓库形态: physical(实体仓) / virtual(虚拟仓)
    category: Mapped[str] = mapped_column(db.String(20), default='physical')

    # 2. 地理位置: domestic(国内) / overseas(海外)
    location_type: Mapped[str] = mapped_column(db.String(20), default='domestic')

    # 3. 管理模式: self(自营) / third_party(三方)
    # 自营=使用本系统WMS流程; 三方=仅通过API交互
    ownership_type: Mapped[str] = mapped_column(db.String(20), default='self')

    status: Mapped[str] = mapped_column(db.String(20), default='active')

    # 业务标签（用于细分监管类型，如: bonded/fba/standard）
    business_type: Mapped[str] = mapped_column(db.String(30), default='standard')

    # 计价币种（用于库存成本与报表换算）
    currency: Mapped[str] = mapped_column(db.String(10), default='USD')

    # JSONB 存储第三方配置，避免频繁改表
    api_config: Mapped[dict] = mapped_column(JSONB, nullable=True)

    # 数组类型存储父级ID
    parent_warehouse_ids: Mapped[list[int]] = mapped_column(ARRAY(db.Integer),
nullable=True)

    # 物理属性
    capacity: Mapped[float] = mapped_column(db.Numeric(12, 2), nullable=True)
    # 增加库容体积限制 (m3) - 汽配特性
    max_volume: Mapped[float] = mapped_column(db.Numeric(12, 2), nullable=True)
    timezone: Mapped[str] = mapped_column(db.String(50), default='UTC')
```

```python
class WarehouseLocation(db.Model):
    """库位表 (新增) - 用于实体仓精细化管理"""
    __tablename__ = 'warehouse_locations'

    id: Mapped[int] = mapped_column(primary_key=True)
    warehouse_id: Mapped[int] = mapped_column(db.ForeignKey('warehouses.id'),
index=True)

    # 库位编码 (如: A-01-01-01 区-排-层-位)
    code: Mapped[str] = mapped_column(db.String(50))

    # 库位类型 (storage:存储, pick:拣货, receive:收货, return:退货, stage:暂存)
    type: Mapped[str] = mapped_column(db.String(20), default='storage')

    # 库位属性 (用于上架策略)
    is_locked: Mapped[bool] = mapped_column(default=False)
    allow_mixing: Mapped[bool] = mapped_column(default=False)  # 是否允许混放SKU


class Stock(db.Model):
    """
    库存余额表 (聚合视图)
    注意：对于实体仓，这是仓库级总库存；对于需要精确到库位的场景，需查询 StockDetail (如
有)
    或通过 stock_movements 聚合。
    为简化设计，v1.0 阶段暂不引入独立的 StockDetail 表，均通过 Stock 表管理仓库级库存。
    库位库存管理将在 Phase 2 结合 PDA 引入。
    """
    __tablename__ = 'stocks'
    # 注意：实际生产中应配置 table_args 进行分区

    id: Mapped[int] = mapped_column(primary_key=True)
    sku: Mapped[str] = mapped_column(db.String(50), index=True)
    warehouse_id: Mapped[int] = mapped_column(db.ForeignKey('warehouses.id'))

    # 库存数量
    physical_quantity: Mapped[int] = mapped_column(default=0)
    available_quantity: Mapped[int] = mapped_column(default=0)
    allocated_quantity: Mapped[int] = mapped_column(default=0)
    in_transit_quantity: Mapped[int] = mapped_column(default=0)
    damaged_quantity: Mapped[int] = mapped_column(default=0)  # 坏品/待退回库存

    # 批次信息
    batch_no: Mapped[str] = mapped_column(db.String(50), nullable=True)

    # 汽配特性：冗余体积重量数据，用于快速计算运费/库容
    weight: Mapped[float] = mapped_column(db.Float, nullable=True)
    volume: Mapped[float] = mapped_column(db.Float, nullable=True)

    # 乐观锁版本号
    version: Mapped[int] = mapped_column(default=0)


class StockMovement(db.Model):
    """库存单据/流水表：入库、出库、调拨、调整等均在此记录，作为库存结果表的审计来源"""
```

```
    __tablename__ = 'stock_movements'

    id: Mapped[int] = mapped_column(primary_key=True)
    sku: Mapped[str] = mapped_column(db.String(50), index=True)
    warehouse_id: Mapped[int] = mapped_column(db.ForeignKey('warehouses.id'))

    # 关联库位 (新增) - 允许为空，待 Phase 2 启用库位管理后必填
    location_id: Mapped[int] =
mapped_column(db.ForeignKey('warehouse_locations.id'), nullable=True)

    # 单据基础信息
    order_type: Mapped[str] = mapped_column(db.String(30))  #
inbound/outbound/transfer/adjustment
    order_no: Mapped[str] = mapped_column(db.String(50), index=True)
    biz_time: Mapped[datetime] = mapped_column(default=datetime.utcnow)

    # 变更数量（正数=增加库存，负数=减少库存）
    quantity_delta: Mapped[int] = mapped_column()

    # 关联批次与成本信息
    batch_no: Mapped[str] = mapped_column(db.String(50), nullable=True)
    unit_cost: Mapped[Decimal] = mapped_column(db.Numeric(18, 4), nullable=True)
    currency: Mapped[str] = mapped_column(db.String(10), nullable=True)

    # 汇率快照 (新增) - 记录交易发生时的汇率
    exchange_rate: Mapped[Decimal] = mapped_column(db.Numeric(10, 4), default=1.0)

    # 审计字段
    created_by: Mapped[int] = mapped_column(nullable=True)
    created_at: Mapped[datetime] = mapped_column(default=datetime.utcnow)
    status: Mapped[str] = mapped_column(db.String(20), default='confirmed')  #
pending/confirmed/cancelled


class StockDiscrepancy(db.Model):
    """库存差异记录表 (用于第三方仓对账与风控告警)"""
    __tablename__ = 'stock_discrepancies'

    id: Mapped[int] = mapped_column(primary_key=True)
    warehouse_id: Mapped[int] = mapped_column(db.ForeignKey('warehouses.id'))
    sku: Mapped[str] = mapped_column(db.String(50))

    local_qty: Mapped[int] = mapped_column()
    remote_qty: Mapped[int] = mapped_column()

    # 差异强度（比例/金额）
    diff_ratio: Mapped[float] = mapped_column(db.Float, nullable=True)
    diff_amount: Mapped[Decimal] = mapped_column(db.Numeric(18, 4), nullable=True)

    # 状态与处理信息
    status: Mapped[str] = mapped_column(default='pending')  #
pending/resolved/ignored
    discovered_at: Mapped[datetime] = mapped_column(default=datetime.utcnow)
    resolved_at: Mapped[datetime] = mapped_column(nullable=True)
    resolver_id: Mapped[int] = mapped_column(nullable=True)
    resolution: Mapped[str] = mapped_column(db.String(200), nullable=True)  # 以第三
方为准/以本地为准/忽略原因
```

3.2 仓库类型与配置 (v1.3 更新)

仓库通过三个核心维度进行分类，支持多种业务组合：

1. **形态 (`category`)**:
    - `physical`: 实体仓，对应真实的库存存放点。
    - `virtual`: 虚拟仓，逻辑上的库存集合。
2. **地理 (`location_type`)**: `domestic`(国内) / `overseas`(海外)。
3. **权属 (`ownership_type`)**:
    - `self`: 自营。系统负责完整的入库、上架、拣货、出库流程（未来对接PDA）。
    - `third_party`: 三方。系统仅负责推送指令（入库单/出库单）和拉取库存/状态。

典型组合示例:

- **国内自营仓**: `physical` + `domestic` + `self`
- **海外三方仓(谷仓)**: `physical` + `overseas` + `third_party`
- **FBA仓**: `physical` + `overseas` + `third_party` (business_type='fba')
- **销售虚拟仓**: `virtual` + `domestic/overseas` + `self` (用于渠道库存分配/超卖控制)
- **采购虚拟仓**: `virtual` + `domestic` + `self` (用于汇聚采购计划与需求缓冲)

3.3 库存余额与库存单据的关系

- `stocks` 表只保存**当前余额**，用于高频查询与前台展示。
- `stock_movements` 表记录**所有库存变动单据/流水**，作为：
    - 余额重算的来源（按时间序列重放）
    - 审计与责任追踪的依据（谁在什么时间对哪个 SKU 做了什么操作）。

业务上所有出入库相关功能（采购入库、销售出库、调拨、盘点等）均应首先在 `stock_movements` 中落单，再驱动 `stocks` 表更新，避免直接对余额做“黑盒调整”。

4. 模块详细设计 (Modules)

不再拆分为独立的微服务进程，而是作为 `backend/app/api/` 下的独立 Blueprints，并在模块边界上与未来服务化保持一致。

4.1 仓库模块 (Warehouse Blueprint)

路径与职责

- **路径**: `backend/app/api/warehouse/`
- **职责**: 仓库基础信息 CRUD，库位管理，第三方 API 配置管理。
- **Service**: `WarehouseService`

仓库模块需要覆盖：

- 管理 `type` 与 `business_type` 等仓库级业务属性。
- **(新增) 库位管理**: 提供 `WarehouseLocation` 的增删改查接口，支持批量导入库位。
- 管理 `currency` 与跨时区 `timezone`。
- 维护第三方仓/平台相关的 `api_config`。

4.2 库存模块 (Stock Blueprint)

- **路径**: `backend/app/api/stock/`
- **职责**: SKU 库存查询、库存余额维护、库存单据（流水）查询。
- **关键逻辑**:
  - **乐观锁控制**: 更新库存时必须携带版本号。

```
# UPDATE stocks SET qty=new_qty, version=version+1 WHERE id=1 AND
version=old_ver
count = Stock.query.filter_by(id=1, version=old_ver).update(...)
if count == 0: raise ConcurrencyError("库存已被修改，请重试")
```

  - **库存单据驱动余额**：所有入库/出库/调拨/调整操作以 `StockMovement` 为主表，`stocks` 仅作为冗余余额表。
  - **汽配特性支持**:
    - **通用件逻辑**: 库存查询接口需预留 `include_interchange` 参数，未来对接产品中心获取通用件关系。
    - **Kit 处理**: 明确区分 **虚拟Kit** (Sales Kit, 动态计算子件库存) 与 **物理Kit** (Assembly Kit, 预包装成品，有独立库存记录)。

4.3 出入库单据模块 (Inventory Order Blueprints)

为避免在库存视图中堆积过多业务逻辑，出入库单据以独立 Blueprint 形式存在：

- **路径示例**：
  - 入库：`backend/app/api/inbound/`
  - 出库：`backend/app/api/outbound/`
  - 调拨：`backend/app/api/transfer/`
  - 盘点/调整：`backend/app/api/adjustment/`
- **统一职责**：
  - 定义各类库存单据的生命周期（创建 → 审核 → 执行 → 完成/取消）
  - 在单据状态流转过程中，写入 `stock_movements` 并驱动 `stocks` 表更新
  - **(新增)** 记录汇率快照：在写入 `stock_movements` 时，记录当前仓库币种对应的汇率。

4.4 虚拟仓模块 (Virtual Warehouse Blueprint)

- **路径**: `backend/app/api/virtual/`
- **职责**: 作为供应链供需两端的**业务缓冲区**，统一管理销售分配与采购需求。
- **核心场景**:
  1. **销售端缓冲 (Outbound Buffer)**:
     - **场景**: 将物理库存分配给不同渠道/团队（如 Amazon, 独立站）。
     - **逻辑**: `虚拟库存 = 物理库存 * 分配规则`。
     - **预售/超卖支持**: 通过配置规则实现（如：`Allow_Oversell = True` 或 `Virtual_Stock = Physical + In_Transit`）。不需要独立的物理表结构，而是通过规则引擎动态计算可售量。
  2. **采购端缓冲 (Inbound Buffer)**:
     - **场景**: 汇聚各团队/渠道的采购需求，形成“计划库存”。
     - **逻辑**: 不同团队根据各自的销售预测策略（如：30天销量平均、季节性系数），计算出“建议采购量”，并将此数值写入对应的“采购虚拟仓”。
     - **作用**:
       - **采前/产中可视化**: 聚焦于从“产生需求”到“入国内仓”之前的数据盲区。让运营能直观看到“正在采购中”或“供应商生产中”的数量。
       - **计划协同**: 运营不再需要翻阅采购合同，直接在系统中看到“未来的货”，从而制定早期的销售策略（预热/测款）。

        - (注：海运/空运的“在途库存”通常明确指向目的实体仓，不归属于此虚拟仓范围)。

- **数据模型调整**:
    - 虚拟仓不再区分复杂的“独立模式”，统一视为**规则驱动的逻辑仓**。
    - 对于采购虚拟仓，其“库存”实际上是 **需求量 (Demand)** 或 **计划量 (Planned Qty)**，在数据上复用 `Stock` 表结构，但 `type` 标记为 `planning`。

4.5 同步模块 (Sync Blueprint & Tasks)

- **路径**: `backend/app/api/sync/`
- **职责**: 触发同步任务、查看同步日志、处理库存差异与告警。
- **差异处理流程**:
    1. **Fetch**: Celery Worker 拉取第三方仓库存数据。
    2. **Compare**: 对比本地库存，计算差异值。
    3. **Record**: 如果有差异，不直接覆盖，而是插入 `StockDiscrepancy` 表。
    4. **Resolve**: 运营人员在前端查看差异单，选择"以第三方为准"(生成调整单)或"忽略"，并记录处理人和结论。

此外需要在同步模块中：

- 按**渠道/平台**维度配置同步策略（全量/增量、频率、安全库存缓冲系数等）
- 对高风险差异（按比例/金额阈值）触发告警（邮件/IM），供风控与运营关注

5. 前端架构设计

5.1 目录结构

```
frontend/apps/web-antd/src/views/
├── warehouse/
│   ├── list/           # 仓库列表
│   ├── detail/         # 仓库详情 (含库位管理)
│   └── location/       # 库位列表/打印
├── stock/
│   ├── overview/       # 库存总览
│   └── adjustment/     # 库存调整
└── virtual/            # 虚拟仓配置
```

5.2 Vxe-Table 使用规范

```
import { useVbenVxeGrid } from '#/adapter/vxe-table';

const [Grid, gridApi] = useVbenVxeGrid({
  gridOptions: {
    columns: [
      { field: 'sku', title: 'SKU' },
      { field: 'physical_quantity', title: '物理库存' }
    ],
    proxyConfig: {
      ajax: {
        query: async ({ page }) => {
          return await getStockList({
            page: page.currentPage,
            per_page: page.pageSize
          });
        }
      }
    }
  }
});
```

6. API 接口详细规范

6.1 视图编写规范 (Python MethodView)

所有接口必须遵循 `code/message/data` 结构，并使用 APIFlask 的装饰器。

示例：获取仓库列表

```
from apiflask.views import MethodView
from apiflask import APIBlueprint
from app.schemas import WarehouseSchema, PaginationQuerySchema, PaginationSchema
from app.services import warehouse_service

bp = APIBlueprint('warehouse', __name__, url_prefix='/warehouses')

class WarehouseListAPI(MethodView):
    @bp.auth_required(auth)
    @bp.doc(summary='获取仓库列表')
    @bp.input(PaginationQuerySchema, location='query', arg_name='query')
    @bp.output(PaginationSchema(WarehouseSchema))  # 自动包装为 {code:0, data:
{items:[], total:..}}
    def get(self, query):
```

```
        """支持分页和搜索"""
        pagination = warehouse_service.get_list(
            page=query['page'],
            per_page=query['per_page'],
            keyword=query.get('q')
        )
        return pagination

bp.add_url_rule('', view_func=WarehouseListAPI.as_view('list'))
```

示例：调整库存

```
class StockAdjustAPI(MethodView):
    @bp.auth_required(auth)
    @bp.doc(summary='库存调整')
    @bp.input(StockAdjustSchema, arg_name='data')
    @bp.output(StockSchema)
    def post(self, data):
        """入库/出库/盘点"""
        try:
            stock = stock_service.adjust(
                sku=data['sku'],
                warehouse_id=data['warehouse_id'],
                quantity=data['quantity'],
                type=data['type'],
                user=current_user
            )
            return stock  # 框架自动序列化并在外层包裹 data
        except BusinessError as e:
            raise e
```

7. 安全设计

- **认证**: 使用 `flask-jwt-extended` 实现 Bearer Token 认证。
- **API 权限**: 使用 `@permission_required('warehouse:view')` 装饰器。
- **数据权限**: 在 Service 层调用 `DataPermissionFilter.apply(query, 'warehouse')` 自动注入 SQL 过滤条件。

8. 性能优化策略

8.1 数据库优化

- **SQLAlchemy 预加载**: 严禁 N+1 查询，使用 `selectinload` 加载虚拟仓规则。
- **分区表**: `stocks` 表按 `warehouse_id` 进行 List 分区。

8.2 缓存策略 (Redis)

- **键命名规范**: `wms:stock:{sku}:{warehouse_id}`。
- **更新策略**: 数据库更新成功后，使用 `Del Cache` 模式（删除缓存），下次读取时重建。

8.3 异步处理

- **Celery 配置**: 参考标准 Flask-Celery 配置。
- **场景**: 第三方库存同步、大批量库存调拨、月度报表生成。

9. 部署方案

9.1 Dockerfile (Python)

```
# 后端构建
FROM python:3.10-slim as builder
WORKDIR /app
COPY requirements.txt .
RUN pip wheel --no-cache-dir --no-deps --wheel-dir /app/wheels -r requirements.txt

# 运行时镜像
FROM python:3.10-slim
WORKDIR /app
COPY --from=builder /app/wheels /wheels
COPY --from=builder /app/requirements.txt .
RUN pip install --no-cache /wheels/*
COPY . .

# 启动命令
CMD ["gunicorn", "-c", "gunicorn_config.py", "run:app"]
```

9.2 Docker Compose 配置

```yaml
version: '3.8'
services:
  backend:
    build: ./backend
    restart: always
    environment:
      - DATABASE_URL=postgresql://user:pass@postgres:5432/db
      - REDIS_URL=redis://redis:6379/0
    volumes:
      - ./backend:/app
    ports:
      - "5000:5000"
    depends_on:
      - redis

  celery_worker:
    build: ./backend
    command: celery -A app.celery_utils.celery worker --loglevel=info
    environment:
      - DATABASE_URL=postgresql://user:pass@postgres:5432/db
      - REDIS_URL=redis://redis:6379/0
    depends_on:
      - redis
      - backend

  redis:
    image: redis:7-alpine
```

9.3 多环境与高可用规划

- **多环境配置**：通过环境变量与配置文件组合区分 `dev/staging/prod`，包含：数据库、Redis、日志等级、外部 API 地址等。
- **数据库高可用**：
  - 采用云厂商提供的主从/高可用实例，启用自动备份
  - 为读多写少场景预留读写分离能力（应用侧通过 SQLAlchemy 绑定不同 Engine）
- **Redis 高可用**：
  - 根据业务量选择哨兵模式或云 Redis 服务
  - 明确持久化策略（RDB/AOF），保证同步任务与异步任务数据安全。

10. 测试策略

10.1 单元测试 (pytest)

放弃 JUnit，采用 Python 社区标准的 `pytest`。

```
# tests/unit/test_stock_service.py
def test_adjust_stock_increase(db_session, stock_factory):
    # Arrange
    stock = stock_factory(sku='TEST001', quantity=100)

    # Act
    service.adjust(stock.sku, 10, 'increase')

    # Assert
    assert stock.quantity == 110
```

10.2 集成测试

使用 `pytest-flask` 提供的 `client` fixture 进行 API 测试。

```
def test_create_warehouse_api(client, admin_token):
    resp = client.post('/api/v1/warehouses',
        json={'code': 'SZ01', 'name': '深圳仓'},
        headers={'Authorization': f'Bearer {admin_token}'}
    )
    assert resp.status_code == 201
    assert resp.json['code'] == 0
    assert resp.json['data']['code'] == 'SZ01'
```

11. 监控告警

- **日志**: 使用 Python `logging` 模块输出 JSON 格式日志，经由 Logtail 采集到阿里云 SLS。
- **应用监控**: 集成 Sentry 或 阿里云 ARMS (Python探针) 进行异常追踪。

12. 移动端/PDA 支持 (技术预备 - Phase 2)

注意: 本章节为技术预备内容，不纳入本次开发迭代（Phase 1）。

12.1 需求概述

为提升仓库现场作业效率，规划引入 PDA (Android 手持设备) 支持。主要用于：

- **入库上架**: 扫描 SKU -> 扫描库位 -> 确认数量。
- **出库拣货**: 扫描拣货单 -> 路径指引 -> 扫描 SKU -> 扫描库位。
- **盘点**: 盲盘/明盘。

12.2 技术方案

- **应用形态**: 独立 H5 应用 / Uni-app / Flutter (待定)，嵌入 Vben Admin 的 Mobile 视图或独立 APK。
- **API 设计**:
  - 需提供专用的轻量级 API (`/api/v1/pda/...`)。
  - 减少数据传输量，仅返回必要字段。
  - 强化扫描纠错能力（如：扫描条码不匹配时立即报错）。
- **交互流程**:
  - 以“单据”为中心，而非“查询”为中心。
  - 必须支持离线/弱网模式（本地缓存操作，网络恢复后上传）。

12.3 待定事项

- PDA 设备选型与屏幕分辨率适配。
- 蓝牙打印机对接方案。