非原子区块链交易识别网站 - 项目最终报告

2025 年 12 月 25 日

1 项目概述

本项目是一个用于识别和分析不同区块链平台间套利机会的网站系统，通过收集并分析来自 **Uniswap V3** 和 **Binance** 等平台的交易数据，识别潜在的套利机会，并提供可视化分析界面。系统主要包含价格对比、套利机会识别、以及详细的统计分析功能。

2 技术架构

本项目采用微服务架构，主要包括以下组件：

- **前端 (Frontend)**：使用 **React** 构建，提供用户界面及数据可视化。
- **后端 (Backend)**：使用 **FastAPI** 提供 RESTful API 接口。
- **数据库 (Database)**：采用 **PostgreSQL** 作为关系型数据库，用于存储交易和套利数据。
- **反向代理 (Reverse Proxy)**：使用 **Nginx** 作为反向代理，负责请求路由。
- **数据获取 (Data Fetching)**：使用 **Python** 脚本，用于定期从各平台 API 获取数据。

3 系统界面展示

本节展示了系统的主要界面截图，展示了系统的用户交互功能。

3.1 价格看板界面

价格看板是系统的主要数据展示界面，提供多个平台的价格对比功能。下图展示了价格看板的主界面：

图 1: 价格看板主界面

如图所示，价格看板提供了Uniswap和Binance的价格对比，并支持交互式操作，包括缩放、拖拽和悬停查看详细信息。

3.2 价格数据图表展示

系统提供了多种数据展示方式，用户可以选择不同的数据源进行查看：

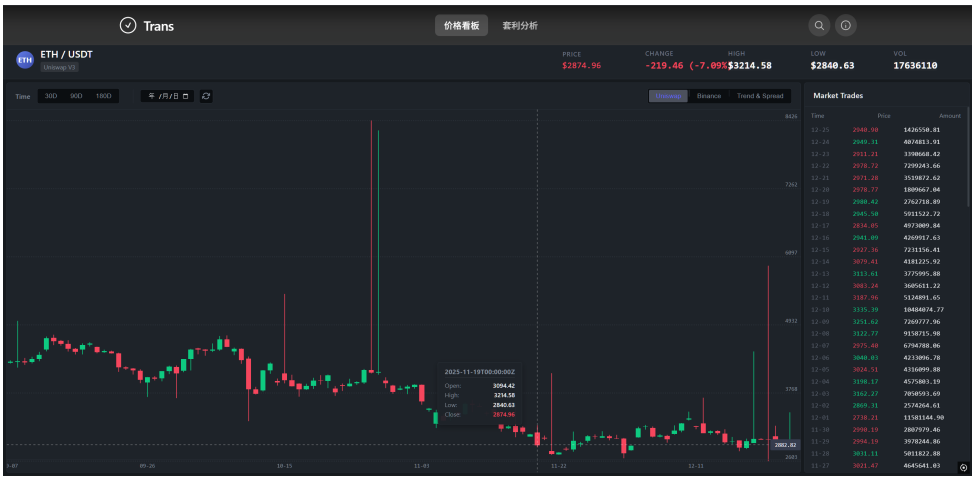

图 2: Uniswap价格数据图表

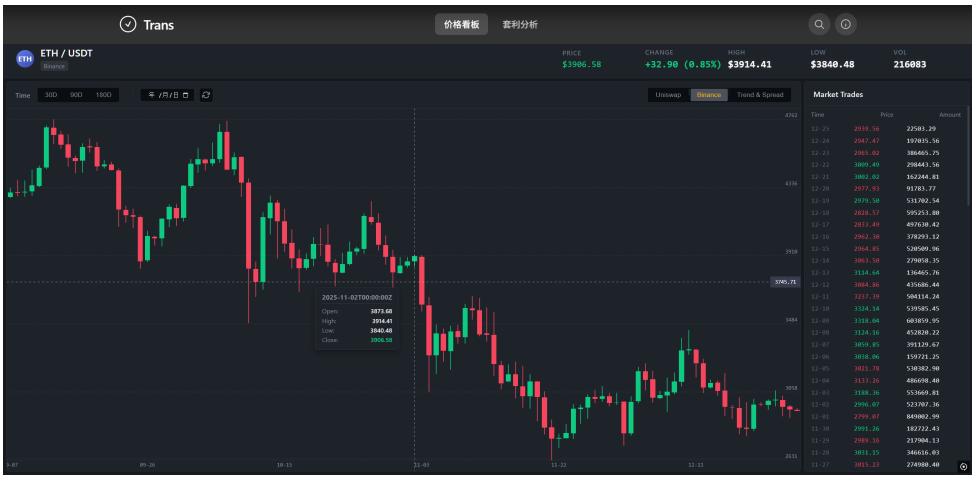

图 3: Binance价格数据图表

图 4: 价格数据汇总视图

3.3 套利分析界面

套利分析界面是系统的核心功能模块，用于识别和分析套利机会：

图 5: 套利分析预览界面

图 6: 套利机会可视化

图 7: 套利行为分析

4 系统组件详细设计

4.1 后端设计

后端采用 FastAPI 框架，主要包含以下模块：

4.1.1 数据模型

后端定义了四个主要的数据模型：

- **UniswapSwap**：存储 Uniswap V3 的 Swap 事件，包含交易哈希、时间戳、价格、数量、流动性等字段。
- **BinanceTrade**：存储 Binance 的成交数据，包含时间戳、价格、数量、开盘价、收盘价等字段。
- **ArbitrageOpportunity**：存储识别出的套利机会，包含时间戳、价格、利润、利润率、方向等字段。
- **ArbitrageOpportunityMinute**：按分钟预计算的套利机会，用于识别潜在的套利机会。

4.1.2 API 接口

后端提供以下主要 API 接口：

- `GET /api/health`：健康检查接口，验证后端服务是否正常运行。

- `GET /api/db-check`：数据库连接检查接口。
- `GET /api/price-data`：获取 Uniswap V3 和 Binance 的价格数据，按天聚合为 OHLC（开高低收）格式。
- `GET /api/arbitrage/statistics`：返回预计算的非原子套利统计数据。
- `GET /api/arbitrage/behaviors`：分页返回识别出的套利行为，支持最小利润过滤和排序。
- `GET /api/arbitrage/opportunities`：获取预计算的套利机会列表（按分钟），支持最小利润率过滤和时间范围筛选。

4.2 前端设计

前端使用 React 框架，主要包含以下页面和组件：

4.2.1 价格看板 (PriceDashboard.js)

价格看板页面提供以下功能：

- 通过 K 线图展示来自多个平台的价格数据。
- 支持切换数据源（Uniswap、Binance 或对比视图）。
- 提供缩放、拖拽、Hover 提示等交互功能。
- 显示关键统计指标，如最新成交价、价格区间、价差、成交量等。
- 支持后端数据和模拟数据切换（通过环境变量 `REACT_APP_USE_BACKEND` 控制）。

4.2.2 套利分析 (ArbitrageAnalysis.js)

套利分析页面提供以下功能：

- 显示套利统计信息（总机会数、总利润、平均利润率）。
- 提供套利行为列表，支持分页、筛选和排序。
- 时间轴可视化展示套利机会。
- 套利方向分布图表（CEX→DEX / DEX→CEX）。
- 累计利润轨迹图。

4.3 数据获取与处理

系统包含多个数据获取和处理脚本：

- `fetch_data`：从 Uniswap V3 和 Binance API 获取交易数据。
- `compute_opportunities`：分析分钟级套利机会。
- `compute_arbitrage`：识别非原子套利候选。
- `reset_recompute.sh`：一键重置并全量重算脚本。

5 关键代码分析

本节分析系统中的关键代码实现，包括后端API实现和前端交互逻辑。

5.1 后端价格数据API实现

以下是后端获取价格数据的API实现，这是系统的核心功能之一：

Listing 1: Price Data API Implementation

```python
@app.get("/api/price-data")
def get_price_data(
    start_date: Optional[date] = Query(
        None, description="Start date in YYYY-MM-DD format (default returns last 30 days)"
    ),
    end_date: Optional[date] = Query(
        None, description="End date in YYYY-MM-DD format (default is today, UTC)"
    ),
    db: Session = Depends(get_db)
):
    """
    Signature: `GET /api/price-data`

    Description:
    Get Uniswap V3 and Binance price data aggregated by day into OHLC format,
    and provide displayTime/id fields directly usable by frontend charts.

```

```
    Parameters:
    - `start_date` (date, optional): Start date in "YYYY-MM-DD" format, default = 29 days before end_date
    - `end_date` (date, optional): End date in "YYYY-MM-DD" format, default = today (UTC)
    - `db` (Session): Database session provided via dependency injection.

    Returns:
    - `dict`: JSON object containing uniswap and binance price data in format:
      {
        "uniswap": [
          {
            "timestamp": "2025-09-01T00:00:00Z",
            "open": 2500.0,
            "high": 2515.0,
            "low": 2490.0,
            "close": 2505.0,
            "volume": 100.0
          }
        ],
        "binance": [...]
      }
    """
    today_utc = datetime.now(timezone.utc).date()
    resolved_end = end_date or today_utc
    resolved_start = start_date or (resolved_end - timedelta(days=DEFAULT_PRICE_WINDOW_DAYS - 1))
    if resolved_start > resolved_end:
        raise HTTPException(status_code=400, detail="start_date must not be after end_date")

    start_dt = _ensure_utc(datetime.combine(resolved_start, time.min))
    end_dt = _ensure_utc(datetime.combine(resolved_end, time.max))

    uniswap_ohlc = _daily_ohlcv(
        db,
```

```
        models.UniswapSwap,
        func.abs(models.UniswapSwap.amount1),
        start_dt,
        end_dt,
    )
    binance_ohlc = _daily_ohlcv(
        db,
        models.BinanceTrade,
        models.BinanceTrade.quantity,
        start_dt,
        end_dt,
    )

    return {
        "uniswap": uniswap_ohlc,
        "binance": binance_ohlc
    }
```

In this API implementation, the `get_price_data` function is responsible for retrieving price data from both platforms and aggregating the raw transaction data into OHLC (Open, High, Low, Close) format via the `_daily_ohlcv` function. The function first processes the date parameters, then retrieves data for Uniswap and Binance separately, and finally returns the result in JSON format.

5.2 套利统计API实现

套利统计API是系统的另一个核心功能，用于计算和返回套利机会的统计数据：

Listing 2: Arbitrage Statistics API Implementation

```
@app.get("/api/arbitrage/statistics")
def get_arbitrage_statistics(db: Session = Depends(get_db)):
    """
    Signature: `GET /api/arbitrage/statistics`

    Description:
    Returns pre-calculated non-atomic arbitrage statistics,
       ↪ directly reading the summary from the
       ↪ arbitrage_opportunities table.
    """
    total_opportunities = db.query(func.count(models.
       ↪ ArbitrageOpportunity.id)).scalar() or 0
```

```
    total_profit = (
        db.query(func.coalesce(func.sum(models.
            ↪ ArbitrageOpportunity.profit), 0)).scalar() or 0.0
    )
    average_profit_rate = (
        db.query(func.coalesce(func.avg(models.
            ↪ ArbitrageOpportunity.profit_rate), 0)).scalar() or
            ↪ 0.0
    )
    return {
        "total_opportunities": int(total_opportunities),
        "total_profit": float(total_profit),
        "average_profit_rate": float(average_profit_rate * 100),
    }
```

This API implementation uses SQLAlchemy to query statistical data from the arbitrage opportunity database, including total opportunities, total profit, and average profit rate. The query uses SQL aggregate functions (COUNT, SUM, AVG) to calculate the statistics.

5.3 前端套利分析组件实现

前端套利分析组件是系统数据展示的核心部分，以下是一些关键代码段：

Listing 3: 套利分析组件关键代码

```
const fetchStatistics = async () => {
  try {
    const response = await fetch(`${API_BASE_URL}/arbitrage/
        ↪ statistics`);
    if (!response.ok) {
      throw new Error('Failed to fetch statistics');
    }
    const data = await response.json();
    setStatistics({
      totalOpportunities: data.total_opportunities || 0,
      totalProfit: data.total_profit || 0,
      averageProfitRate: data.average_profit_rate || 0
    });
  } catch (err) {
    console.error('Error fetching statistics:', err);
```

```
    setError('Failed to fetch statistics, please try again later');
  }
};

const fetchBehaviors = async () => {
  try {
    setLoading(true);
    const params = new URLSearchParams({
      page: currentPage.toString(),
      page_size: pageSize.toString(),
      sort_by: filters.sortBy,
      sort_order: filters.sortOrder
    });

    if (filters.minProfit) {
      params.append('min_profit', filters.minProfit);
    }

    const response = await fetch(`${API_BASE_URL}/arbitrage/behaviors?${params}`);
    if (!response.ok) {
      throw new Error('Failed to fetch arbitrage behavior list');
    }
    const data = await response.json();
    setBehaviors(data.behaviors || []);
    setTotalPages(data.total_pages || 1);
    setError(null);
  } catch (err) {
    console.error('Error fetching arbitrage behavior list:', err);
    setError('Failed to fetch arbitrage behavior list, please try again later');
    setBehaviors([]);
  } finally {
    setLoading(false);
  }
};
```

在前端组件中，通过fetch函数调用后端API，使用React的useState和useEffect钩子

管理组件状态。代码实现了错误处理、参数构建和数据更新功能。

5.4 数据聚合函数实现

后端的数据聚合函数是系统数据处理的核心：

Listing 4: Data Aggregation Function Implementation

```python
def _daily_ohlcv(
    db: Session,
    model,
    volume_expr,
    start_dt: datetime,
    end_dt: datetime,
) -> List[Dict[str, float]]:
    """
    Aggregates table data containing price/timestamp into daily OHLCV format.
    Additionally returns id (row number) and displayTime for compatibility with existing frontend structure.
    """
    grouped = (
        db.query(
            cast(model.timestamp, Date).label("date"),
            func.min(model.price).label("low"),
            func.max(model.price).label("high"),
            func.sum(volume_expr).label("volume"),
        )
        .filter(model.timestamp >= start_dt, model.timestamp <= end_dt)
        .group_by(cast(model.timestamp, Date))
        .order_by(cast(model.timestamp, Date))
        .all()
    )

    results: List[Dict[str, float]] = []
    for idx, row in enumerate(grouped, start=1):
        current_date = row.date
        first_trade = (
            db.query(model)
            .filter(cast(model.timestamp, Date) == current_date)
```

```
            .order_by(model.timestamp.asc())
            .first()
        )
        last_trade = (
            db.query(model)
            .filter(cast(model.timestamp, Date) == current_date)
            .order_by(model.timestamp.desc())
            .first()
        )
        if not first_trade or not last_trade:
            continue

        ts = _ensure_utc(datetime.combine(current_date, time.min)
           ↪ )
        results.append(
            {
                "id": idx,
                "timestamp": ts.isoformat().replace("+00:00", "Z"
                   ↪ ),
                "displayTime": ts.strftime("%m-%d"),
                "open": float(first_trade.price or 0.0),
                "high": float(row.high or first_trade.price or
                   ↪ 0.0),
                "low": float(row.low or first_trade.price or 0.0)
                   ↪ ,
                "close": float(last_trade.price or 0.0),
                "volume": float(row.volume or 0.0),
            }
        )
    return results
```

This function implements the aggregation of raw transaction data in the database into OHLCV format, including Open, High, Low, Close prices, and Volume. The function uses SQL's GROUP BY and aggregate functions to aggregate the data, and retrieves the opening and closing prices through additional queries.

6 部署方案

项目使用 Docker Compose 进行服务编排，包含以下服务：

- **db**：PostgreSQL 数据库服务，使用持久化卷存储数据。
- **backend**：FastAPI 后端服务，依赖数据库健康检查。
- **frontend**：React 前端服务，支持热重载。
- **nginx**：Nginx 反向代理，作为统一入口。

环境变量配置通过 .env 文件管理，包括数据库连接信息、API 密钥等。

7 技术栈和依赖

7.1 后端依赖 (requirements.txt)

- fastapi
- uvicorn[standard]
- sqlalchemy
- psycopg2-binary
- python-dotenv
- requests

7.2 前端依赖 (package.json)

- react
- react-dom
- react-router-dom
- react-scripts
- @testing-library/jest-dom
- @testing-library/react
- @testing-library/user-event

8 系统特点与创新

8.1 数据处理能力

系统能够处理来自多个平台的高频交易数据，并将其统一为标准的 OHLC 格式，便于对比分析。

8.2 可视化功能

系统提供了丰富的可视化功能，包括价格对比图、套利机会时间轴、方向分布图和累计利润轨迹图。

8.3 套利算法

套利识别算法包含四个步骤：

1. 数据对齐与清洗：同步 Uniswap 与 Binance 的时间序列，剔除异常值。
2. 价差与成本计算：计算 DEX/CEX 实时价差，并叠加滑点、手续费、Gas 等成本。
3. 机会筛选与方向判定：当净价差超过阈值时，标注套利方向并记录信号强度。
4. 收益估算与跟踪：以成交量与时间窗口为约束，估算潜在利润并监控机会的持续时间。

8.4 灵活性与可扩展性

系统设计具有良好的灵活性，支持：

- 后端/模拟数据切换
- 多种时间范围和粒度选择
- 可配置的筛选条件
- 模块化的组件设计

9 项目总结

本项目成功实现了非原子区块链交易识别网站，提供了从数据获取、处理、存储到可视化的完整解决方案。系统架构清晰、功能完善，能够有效识别和分析不同平台间的套利机会。通过现代化的技术栈和良好的架构设计，项目具有良好的可维护性和扩展性。

项目的主要成果包括：

- 完整的前后端分离架构
- 高效的数据处理和存储方案
- 丰富的可视化分析功能
- 灵活的部署方案

未来可以考虑增加更多的交易对支持、更复杂的套利策略以及更高级的分析功能。