A股量化交易框架完整设计文档 (QTrader Framework)

框架简称: QTrader (Quantitative Trading Framework for A-shares)
版本: 5.0.0 (完整可交付版本)
最后更新: 2025年10月

目录

---

1. 系统概述

1.1 框架定位

QTrader是一个**专注于交易逻辑、数据完全解耦、支持回测与模拟盘**的A股量化交易框架。框架的核心理念是：

- **框架只管交易**：时间推进、订单管理、撮合成交、账户更新、绩效统计
- **数据由用户提供**：框架定义数据接口合约，用户实现具体数据获取逻辑
- **策略由用户编写**：框架提供生命周期钩子，用户编写策略逻辑

1.2 核心特性

1. **数据完全解耦**
   - 框架不自带任何数据获取实现
   - 定义清晰的数据接口合约（AbstractDataProvider）
   - 用户根据自己的数据源（API、本地文件、数据库）实现接口
   - 框架仅关心交易必需数据：交易日历、当前价格快照、收盘价、基准价格

2. **回测/模拟盘统一**
   - 同一策略代码可无缝切换回测与模拟盘模式
   - 配置驱动模式切换，无需修改策略代码
   - 数据工具保持一致

3. **精准撮合机制**

- 仅支持限价单和市价单
- 市价单：以ask1/bid1价格成交
- 限价单——智能撮合：
  - **即时订单**：当前时刻可成交时，以ask1/bid1成交
  - **历史订单**：跨时间点成交时，以用户限价成交（避免未来函数）
- 回退机制：无五档数据时使用current_price
- 涨跌停检查：当前价格等于涨跌停价时拒绝
- 用户切换 T+1/T+0
- 限价单智能撮合：每次handle_bar自动检查历史未成交订单

4. **多频率支持**

   - 日频：支持多时间点执行
   - 分钟频：每分钟执行一次
   - Tick频：每3秒执行一次（带时间校准）

5. **丰富的生命周期钩子**

   - initialize：策略初始化
   - before_trading：盘前准备（09:25）
   - handle_bar：盘中执行（14:55，支持多时间点，可自定义）
   - after_trading：盘后处理（15:01）
   - broker_settle：日终结算（15:30，可自定义）
   - on_end：策略结束

6. **实时可视化**

   - Web界面实时展示回测/模拟盘运行状态
   - 权益曲线、持仓详情、订单记录
   - 基于WebSocket的数据推送

7. **状态可持久化**

   - 支持暂停/恢复
   - 每日自动保存
   - 完整状态恢复

1.3 框架不负责的部分

明确声明：框架不提供以下功能

1. **数据获取与存储**：框架不下载数据、不维护数据库
2. **因子计算**：框架不提供任何技术指标或因子库
3. **历史数据查询**：框架不负责获取历史K线数据（这是策略的职责）
4. **选股逻辑**：框架不提供股票筛选工具
5. **实盘下单**：仅支持模拟盘，不对接券商API（用户可在broker_settle中自行对接）

用户需要自行实现：

- 数据提供者（实现AbstractDataProvider接口）
- 策略逻辑（选股、择时、仓位管理、历史数据查询）
- 因子库（如需使用）
- 实盘下单逻辑（如需对接券商API）

---

2. 核心设计理念

2.1 架构原则

2.1.1 关注点分离（Separation of Concerns）

2.1.2 依赖注入（Dependency Injection）

框架不创建数据提供者，而是要求用户在启动时注入：

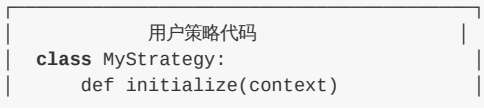
```
# 用户实现数据提供者
data_provider = MyDataProvider(...)

# 注入到引擎
engine = Engine(config_path='configs/backtest.yaml')
engine.run(
    strategy_class=MyStrategy,
    data_provider=data_provider  # 注入
)
```

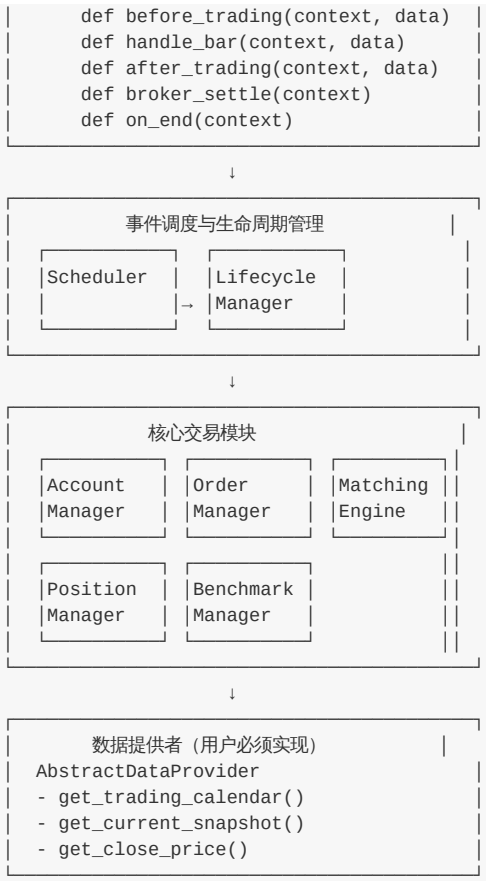
2.1.3 配置驱动（Configuration Driven）

所有行为通过配置文件控制，无需修改代码：

```
mode: backtest  # 或 simulation
frequency: daily  # 或 minute、tick
initial_cash: 1000000
benchmark:
  symbol: "000300"
handle_bar_times:  # 支持多时间点
  - "10:00:00"
  - "14:55:00"
```

2.2 分层架构





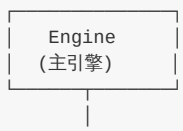
3. 系统架构设计

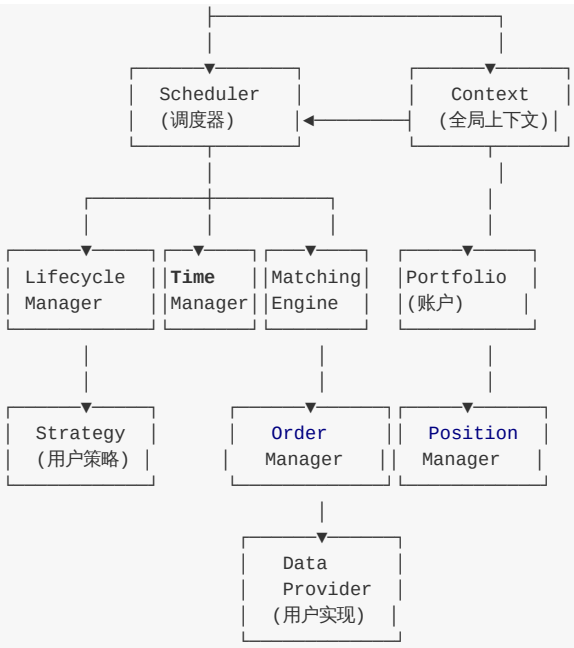
3.1 目录结构

```
qtrader/
├── core/                       # 核心框架
│   ├── __init__.py
│   ├── context.py              # 全局上下文对象
│   ├── engine.py               # 主引擎
│   ├── scheduler.py            # 事件调度器
│   ├── time_manager.py         # 时间管理器
│   ├── lifecycle.py            # 生命周期管理器
│   └── config.py               # 配置管理器
│
├── trading/                    # 交易核心
│   ├── __init__.py
│   ├── account.py              # 账户对象
│   ├── order.py                # 订单对象
│   ├── order_manager.py        # 订单管理器
│   ├── position.py             # 持仓对象
│   ├── position_manager.py     # 持仓管理器
│   ├── matching_engine.py      # 撮合引擎
│   ├── commission.py           # 手续费计算
│   └── slippage.py             # 滑点模型
│
├── benchmark/                  # 基准管理
│   ├── __init__.py
│   └── benchmark_manager.py    # 基准管理器
│
├── data/                       # 数据接口层
│   ├── __init__.py
│   ├── interface.py            # 数据接口合约定义
```

```
│   └── examples/             # 示例实现
│       ├── stock_api_provider.py
│       └── local_csv_provider.py
│
├── strategy/                  # 策略相关
│   ├── __init__.py
│   ├── base.py               # 策略基类
│   ├── context.py            # 策略上下文
│   └── data_proxy.py         # 数据代理
│
├── visualization/             # 实时可视化
│   ├── __init__.py
│   ├── server.py             # Web服务器
│   ├── websocket.py          # WebSocket推送
│   ├── templates/            # HTML模板
│   │   └── dashboard.html
│   └── static/               # 静态资源
│       ├── css/
│       │   └── style.css
│       └── js/
│           └── dashboard.js
│
├── analysis/                  # 分析模块
│   ├── __init__.py
│   ├── performance.py        # 绩效分析
│   ├── risk.py               # 风险分析
│   ├── report.py             # 报告生成
│   └── visualizer.py         # 图表生成
│
├── utils/                     # 工具模块
│   ├── __init__.py
│   ├── logger.py             # 日志工具
│   ├── serializer.py         # 序列化工具
│   └── helpers.py            # 辅助函数
│
├── configs/                   # 配置文件目录
│   ├── default.yaml          # 默认配置
│   ├── backtest.yaml         # 回测配置
│   └── simulation.yaml       # 模拟盘配置
│
├── examples/                  # 示例代码
│   ├── strategies/           # 策略示例
│   │   ├── simple_ma.py
│   │   ├── mean_reversion.py
│   │   └── grid_trading.py
│   └── data_providers/       # 数据提供者示例
│       ├── stock_api_impl.py
│       └── local_csv_impl.py
│
├── tests/                     # 测试代码
│   ├── unit/                 # 单元测试
│   ├── integration/          # 集成测试
│   └── e2e/                  # 端到端测试
│
├── docs/                      # 文档目录
│   ├── user_guide.md         # 用户指南
│   ├── api_reference.md      # API参考
│   └── data_provider_guide.md # 数据提供者开发指南
│
├── setup.py                   # 安装脚本
├── requirements.txt           # 依赖列表
├── README.md                  # 项目说明
└── LICENSE                    # 许可证
```

3.2 组件关系图





4. 核心对象模型

4.1 Context（全局上下文）

Context是框架的核心对象，贯穿整个策略运行过程。它存储了所有运行时状态，并提供了访问各个管理器的接口。

4.1.1 Context设计

```python
# core/context.py

from datetime import datetime
from typing import Dict, Any, Optional
from dataclasses import dataclass, field

@dataclass
class Context:
    """
    全局上下文对象

    这是框架中最核心的对象，存储了策略运行过程中的所有状态信息。
    用户可以通过context访问账户、持仓、订单等信息，也可以存储自定义数据。

    重要说明：
    1. Context对象在策略的整个生命周期中保持存在
    2. user_data字典中的数据会在每日结算后保留到第二天
    3. 某些临时状态（如当日订单列表）会在日终清空
    """

    # ========== 基础运行信息 ==========
    mode: str = 'backtest'  # 运行模式: 'backtest' 或 'simulation'
    strategy_name: str = ''  # 策略名称
    start_date: str = ''  # 回测开始日期（YYYY-MM-DD）
    end_date: str = ''  # 回测结束日期（YYYY-MM-DD）
    current_dt: Optional[datetime] = None  # 当前时间（精确到秒）

    # ========== 频率设置 ==========
    frequency: str = 'daily'  # 运行频率: 'daily', 'minute', 'tick'

    # ========== 核心管理器 ==========
    portfolio: Optional['Portfolio'] = None  # 账户管理器
```

```
    order_manager: Optional['OrderManager'] = None  # 订单管理器
    position_manager: Optional['PositionManager'] = None  # 持仓管理器
    benchmark_manager: Optional['BenchmarkManager'] = None  # 基准管理器

    # ========== 配置信息 ==========
    config: Dict[str, Any] = field(default_factory=dict)  # 完整配置字典

    # ========== 用户自定义数据存储 ==========
    # 这个字典中的数据会持久化到第二天
    user_data: Dict[str, Any] = field(default_factory=dict)

    # ========== 运行状态 ==========
    is_running: bool = False  # 是否正在运行
    is_paused: bool = False  # 是否已暂停

    # ========== 可视化服务 ==========
    visualization_server: Optional['VisualizationServer'] = None

    # ========== 日志 ==========
    logger: Optional['Logger'] = None

    # ========== 数据提供者（用户注入） ==========
    data_provider: Optional['AbstractDataProvider'] = None

    def __post_init__(self):
        """初始化后处理"""
        if self.current_dt is None:
            self.current_dt = datetime.now()

    def get(self, key: str, default: Any = None) -> Any:
        """
        获取用户自定义数据

        Args:
            key: 数据键名
            default: 默认值（如果键不存在）

        Returns:
            对应的值或默认值

        Example:
            ma_short = context.get('ma_short', 5)
        """
        return self.user_data.get(key, default)

    def set(self, key: str, value: Any):
        """
        设置用户自定义数据

        Args:
            key: 数据键名
            value: 数据值

        Example:
            context.set('ma_short', 5)
            context.set('price_history', [])
        """
        self.user_data[key] = value

    def register_data_provider(self, data_provider: 'AbstractDataProvider'):
        """
        注册数据提供者（由引擎调用）

        Args:
            data_provider: 用户实现的数据提供者实例
        """
        self.data_provider = data_provider
```

4.1.2 Context生命周期说明

Context的生命周期贯穿整个策略运行过程：

1. **创建时机**：引擎初始化时创建
2. **存在期间**：从策略开始到结束
3. **销毁时机**：策略运行结束或异常退出

数据持久化规则：

数据类型	是否持久化到第二天	说明
user_data字典	✓ 是	用户自定义数据会保留
账户余额	✓ 是	portfolio.cash会保留
持仓信息	✓ 是	position_manager中的持仓保留
当日订单	× 否	order_manager在日终会清空当日订单
当日成交记录	× 否	但会记录到历史成交列表

用户如何保存自定义数据：

```
def initialize(context):
    # 初始化时设置
    context.set('price_history', [])
    context.set('ma_short', 5)

def handle_bar(context, data):
    # 每次都可以访问
    price_history = context.get('price_history')

    # 修改后保存
    price_history.append(data.current('000001', 'close'))
    context.set('price_history', price_history)

    # 第二天依然可以访问这些数据
```

4.2 Portfolio（账户对象）

Portfolio对象管理账户的现金和总资产。

```
# trading/account.py

from typing import List, Dict

class Portfolio:
    """
    账户对象

    管理账户的现金、总资产等信息。
    """

    def __init__(self, initial_cash: float):
        """
        Args:
            initial_cash: 初始资金
        """
        self.initial_cash = initial_cash  # 初始资金
        self.cash = initial_cash  # 当前可用现金
        self.total_value = initial_cash  # 当前总资产（现金+持仓市值）

        # 历史记录（用于绘制权益曲线）
        self.history: List[Dict] = []
```

```
    def record_history(self):
        """记录当日账户状态（每日结算时调用）"""
        self.history.append({
            'date': self.context.current_dt.strftime('%Y-%m-%d'),
            'cash': self.cash,
            'total_value': self.total_value,
            'returns': (self.total_value - self.initial_cash) / self.initial_cash,
        })

    @property
    def returns(self) -> float:
        """当前收益率"""
        return (self.total_value - self.initial_cash) / self.initial_cash
```

4.3 Order（订单对象）

```
# trading/order.py

from datetime import datetime
from typing import Optional
from enum import Enum

class OrderSide(Enum):
    """订单方向"""
    BUY = 'buy'   # 买入
    SELL = 'sell'  # 卖出

class OrderType(Enum):
    """订单类型"""
    MARKET = 'market'  # 市价单
    LIMIT = 'limit'    # 限价单

class OrderStatus(Enum):
    """订单状态"""
    OPEN = 'open'           # 已提交，未成交
    FILLED = 'filled'       # 已成交
    REJECTED = 'rejected'  # 已拒绝
    EXPIRED = 'expired'    # 已过期（仅限价单）

class Order:
    """
    订单对象

    表示一个交易订单，包含订单的所有信息和状态。

    关键属性：
    - is_immediate: 是否为即时订单（当前时刻提交的）
    - created_bar_time: 订单创建时的bar时间（用于区分即时和历史订单）
    """

    def __init__(
        self,
        symbol: str,
        amount: int,
        side: OrderSide,
        order_type: OrderType,
        limit_price: Optional[float] = None
    ):
        """
        Args:
            symbol: 股票代码（不带后缀）
            amount: 数量（股）
            side: 买卖方向
            order_type: 订单类型
            limit_price: 限价（仅限价单需要）
        """
        from ..utils.helpers import generate_order_id

        self.id = generate_order_id()  # 订单ID
```

```
        self.symbol = symbol
        self.amount = amount
        self.side = side
        self.order_type = order_type
        self.limit_price = limit_price

        # 状态
        self.status = OrderStatus.OPEN

        # 时间
        self.created_time: Optional[datetime] = None  # 订单创建时间
        self.created_bar_time: Optional[datetime] = None  # 订单创建时的bar时间
        self.filled_time: Optional[datetime] = None  # 成交时间

        # 成交信息
        self.filled_price: Optional[float] = None
        self.commission: Optional[float] = None

        # 是否为即时订单（当前bar提交的）
        self.is_immediate: bool = True

    def fill(self, price: float, commission: float, dt: datetime):
        """
        标记订单为已成交

        Args:
            price: 成交价格
            commission: 手续费
            dt: 成交时间
        """
        self.status = OrderStatus.FILLED
        self.filled_price = price
        self.commission = commission
        self.filled_time = dt

    def reject(self, reason: str):
        """
        标记订单为已拒绝

        Args:
            reason: 拒绝原因
        """
        self.status = OrderStatus.REJECTED

    def expire(self):
        """标记订单为已过期"""
        self.status = OrderStatus.EXPIRED

    def mark_as_historical(self):
        """标记订单为历史订单"""
        self.is_immediate = False
```

4.4 Position（持仓对象）

```
# trading/position.py

from datetime import datetime
from typing import Optional

class Position:
    """
    持仓对象

    表示某个股票的持仓信息。

    重要属性：
    - total_amount: 总持仓数量
    - available_amount: 可卖出数量（T+1处理）
    - today_open_amount: 今日买入数量（T+1当日不可卖）
```

```
    """

    def __init__(
        self,
        symbol: str,
        amount: int,
        avg_cost: float,
        current_dt: datetime
    ):
        """
        Args:
            symbol: 股票代码
            amount: 持仓数量
            avg_cost: 平均成本
            current_dt: 建仓时间
        """
        self.symbol = symbol
        self.total_amount = amount  # 总持仓数量
        self.available_amount = 0  # 可卖出数量（T+1处理）
        self.today_open_amount = 0  # 今日买入数量
        self.avg_cost = avg_cost  # 平均成本
        self.current_price: Optional[float] = None  # 当前价格
        self.init_time = current_dt  # 建仓时间
        self.last_update_time = current_dt  # 最后更新时间

    @property
    def market_value(self) -> float:
        """当前市值"""
        if self.current_price is None:
            return 0.0
        return self.total_amount * self.current_price

    @property
    def unrealized_pnl(self) -> float:
        """浮动盈亏"""
        if self.current_price is None:
            return 0.0
        return (self.current_price - self.avg_cost) * self.total_amount

    @property
    def unrealized_pnl_ratio(self) -> float:
        """浮动盈亏率"""
        if self.avg_cost == 0:
            return 0.0
        return (self.current_price - self.avg_cost) / self.avg_cost

    def update_price(self, price: float):
        """
        更新当前价格

        Args:
            price: 最新价格
        """
        self.current_price = price
```

5. 数据接口合约

5.1 设计理念

数据接口合约（Data Interface Contract）是框架与用户数据层之间的唯一桥梁。框架定义清晰的接口规范，用户必须实现这些接口，并在启动时将实例注入到引擎中。

核心原则：

1. **框架不关心数据来源**：API、本地文件、数据库、内存，都可以
2. **框架只关心接口规范**：只要实现了AbstractDataProvider接口即可

3. **用户完全控制数据逻辑**：如何获取、如何缓存、如何优化，完全由用户决定
4. **框架不负责历史数据**：历史K线数据由用户策略自行获取，框架只需要当前价格快照

5.2 AbstractDataProvider（抽象数据提供者）

这是框架定义的唯一数据接口，用户**必须（MUST）**实现这个抽象基类。

```python
# data/interface.py

from abc import ABC, abstractmethod
from typing import List, Dict, Optional
from datetime import datetime

class AbstractDataProvider(ABC):
    """
    抽象数据提供者

    这是框架与数据层之间的接口合约。用户必须实现这个抽象基类，
    并在启动引擎时将实例注入。

    框架运行所需的所有数据都通过这个接口获取：
    1. 交易日历
    2. 当前价格快照（用于撮合）
    3. 收盘价（用于结算和基准更新）

    重要说明：
    - 框架不负责获取历史K线数据，这是策略的职责
    - 用户可以使用任何数据源（API、本地文件、数据库等）
    - 用户可以自行实现缓存策略以优化性能
    - 框架保证不会重复请求同一时间点的数据
    """

    @abstractmethod
    def get_trading_calendar(self, start: str, end: str) -> List[str]:
        """
        获取交易日历

        返回指定时间范围内的所有交易日列表。

        Args:
            start: 开始日期，格式 'YYYY-MM-DD'
            end: 结束日期，格式 'YYYY-MM-DD'

        Returns:
            交易日列表，格式 ['YYYY-MM-DD', 'YYYY-MM-DD', ...]
            必须按时间升序排列

        Example:
            >>> provider.get_trading_calendar('2023-01-01', '2023-01-31')
            ['2023-01-03', '2023-01-04', '2023-01-05', ...]

        注意：
            - 返回的日期必须都是交易日
            - 建议在实现时进行本地缓存以提高性能
            - 框架在初始化时会调用一次以获取完整交易日历
        """
        pass

    @abstractmethod
    def get_current_snapshot(
        self,
        symbol: str,
        dt: datetime,
        frequency: str
    ) -> Optional[Dict]:
        """
        获取当前时刻的价格快照

        返回指定标的在指定时刻的完整价格信息（一次性返回所有必需数据）。
        这个接口用于撮合成交，必须包含：当前价格、五档数据、涨跌停价、停牌状态。
```

```
        Args:
            symbol: 股票代码（不带后缀），如 '000001'
            dt: 查询时间（精确到秒）
            frequency: 频率，'daily'、'minute'、'tick'

        Returns:
            价格快照字典，格式：
            {
                # 基础价格信息（必需）
                'current_price': 10.5,    # 当前价格

                # 五档数据（可选，用于精确撮合）
                'ask1': 10.51,            # 卖一价
                'bid1': 10.49,            # 买一价
                # 注意：如果没有五档数据，ask1和bid1可以为None
                #       撮合引擎会自动使用current_price

                # 涨跌停价（必需）
                'high_limit': 11.55,      # 涨停价
                'low_limit': 9.45,        # 跌停价

                # 停牌状态（必需）
                'is_suspended': False,    # 是否停牌
            }

            如果无法获取数据（如股票不存在），返回None

        Example:
            >>> provider.get_current_snapshot(
            ...     '000001',
            ...     datetime(2023, 1, 10, 14, 30, 0),
            ...     'minute'
            ... )
            {
                'current_price': 10.5,
                'ask1': 10.51,
                'bid1': 10.49,
                'high_limit': 11.55,
                'low_limit': 9.45,
                'is_suspended': False,
            }

        重要说明：
            - 这个接口一次性返回所有撮合所需数据，避免多次调用
            - current_price是必需的，其他字段如果没有可以为None
            - 框架会优先使用ask1/bid1进行撮合，如果为None则使用current_price
            - 涨跌停价如果为None，框架会跳过涨跌停检查
            - 实现时建议进行短期缓存（如5秒），避免重复请求
        """
        pass

    @abstractmethod
    def get_close_price(self, symbol: str, date: str) -> Optional[float]:
        """
        获取收盘价

        返回指定标的在指定日期的收盘价。
        这个接口用于：
        1. 每日结算时更新持仓市值
        2. 基准管理器更新基准价格

        Args:
            symbol: 股票代码（不带后缀）
            date: 日期，格式 'YYYY-MM-DD'

        Returns:
            收盘价，如果没有数据返回None

        Example:
            >>> provider.get_close_price('000001', '2023-01-10')
            10.65

        注意：
            - 这个接口会在每日收盘后调用
```

```
            - 建议实现时进行缓存
        """
        pass
```

5.3 数据提供者示例实现

5.3.1 基于Stock API的实现

```python
# data/examples/stock_api_provider.py

import os
import json
from typing import List, Dict, Optional
from datetime import datetime, timedelta
from ..interface import AbstractDataProvider

class StockAPIProvider(AbstractDataProvider):
    """
    基于Stock API的数据提供者实现

    这是一个完整的示例实现，展示了如何基于Stock API SDK
    实现AbstractDataProvider接口。

    特点：
    1. 交易日历本地缓存（自动增量更新）
    2. 价格数据短期缓存（5秒）
    3. 完整的错误处理
    """

    def __init__(self, api_client, cache_dir: str = '.cache'):
        """
        Args:
            api_client: Stock API客户端实例
            cache_dir: 缓存目录
        """
        self.api_client = api_client
        self.cache_dir = cache_dir
        os.makedirs(cache_dir, exist_ok=True)

        # 交易日历缓存
        self.calendar_file = os.path.join(cache_dir, 'calendar.json')
        self.calendar_data = self._load_calendar()
        self._update_calendar()

        # 价格数据缓存（内存）
        self._price_cache: Dict[str, Dict] = {}
        self._cache_time: Dict[str, datetime] = {}

    def _load_calendar(self) -> dict:
        """加载本地交易日历缓存"""
        if os.path.exists(self.calendar_file):
            with open(self.calendar_file, 'r') as f:
                return json.load(f)
        return {'trading_days': [], 'last_update': None}

    def _save_calendar(self):
        """保存交易日历到本地"""
        with open(self.calendar_file, 'w') as f:
            json.dump(self.calendar_data, f, indent=2)

    def _update_calendar(self):
        """自动更新交易日历"""
        last_update = self.calendar_data.get('last_update')

        # 如果从未更新，或距离上次更新超过1天，则更新
        should_update = (
            last_update is None or
            datetime.now() - datetime.fromisoformat(last_update) > timedelta(days=1)
        )
```

```
        if should_update:
            try:
                # 确定更新范围
                if self.calendar_data['trading_days']:
                    start_date = max(self.calendar_data['trading_days'])
                else:
                    start_date = '2005-01-01'

                end_date = f"{datetime.now().year + 1}-12-31"

                # 调用API
                new_days = self.api_client.calendar.get_trading_days(
                    start_date=start_date,
                    end_date=end_date
                )

                # 合并去重
                all_days = set(self.calendar_data['trading_days'] + new_days)
                self.calendar_data['trading_days'] = sorted(list(all_days))
                self.calendar_data['last_update'] = datetime.now().isoformat()

                # 保存
                self._save_calendar()

                print(f"交易日历更新成功，共 {len(self.calendar_data['trading_days'])} 个交易日")

            except Exception as e:
                print(f"更新交易日历失败: {e}")

    def get_trading_calendar(self, start: str, end: str) -> List[str]:
        """获取交易日历"""
        return [
            day for day in self.calendar_data['trading_days']
            if start <= day <= end
        ]

    def get_current_snapshot(
        self,
        symbol: str,
        dt: datetime,
        frequency: str
    ) -> Optional[Dict]:
        """获取当前价格快照（一次性获取所有必需数据）"""
        # 检查缓存（5秒有效期）
        cache_key = f"{symbol}_{dt.isoformat()}_{frequency}"
        if cache_key in self._price_cache:
            cache_time = self._cache_time.get(cache_key)
            if cache_time and (datetime.now() - cache_time).total_seconds() < 5:
                return self._price_cache[cache_key]

        try:
            snapshot = {}
            date_str = dt.strftime('%Y-%m-%d')

            # 1. 获取价格和五档数据
            if frequency == 'tick':
                # 使用tick接口
                tick_data = self._get_tick_data(symbol, dt)
                if tick_data:
                    snapshot['current_price'] = tick_data.get('current_price')
                    snapshot['ask1'] = tick_data.get('ask1_price')
                    snapshot['bid1'] = tick_data.get('bid1_price')
            else:
                # 使用虚拟K线接口
                kline_data = self._get_vkline_data(symbol, dt, frequency)
                if kline_data:
                    snapshot['current_price'] = kline_data.get('current_price')
                    snapshot['ask1'] = None
                    snapshot['bid1'] = None

            if not snapshot.get('current_price'):
                return None

            # 2. 获取涨跌停价格
```

```
            limit_result = self.api_client.market.get_limit_price(
                code=symbol,
                date=date_str
            )
            if limit_result and symbol in limit_result:
                limit_data = limit_result[symbol]
                snapshot['high_limit'] = limit_data.get('high_limit')
                snapshot['low_limit'] = limit_data.get('low_limit')
            else:
                snapshot['high_limit'] = None
                snapshot['low_limit'] = None

            # 3. 判断停牌
            suspend_result = self.api_client.stock.is_suspended(
                code=symbol,
                date=date_str
            )
            snapshot['is_suspended'] = (
                suspend_result[0].get('is_suspended', False)
                if suspend_result else False
            )

            # 写入缓存
            self._price_cache[cache_key] = snapshot
            self._cache_time[cache_key] = datetime.now()

            return snapshot

        except Exception as e:
            print(f"获取价格快照失败 {symbol} @ {dt}: {e}")
            return None

    def get_close_price(self, symbol: str, date: str) -> Optional[float]:
        """获取收盘价"""
        try:
            result = self.api_client.market.get_kline(
                code=symbol,
                period='1d',
                end_time=date,
                count=1
            )

            if result and symbol in result and len(result[symbol]) > 0:
                return result[symbol][0].get('close')

            return None

        except Exception as e:
            print(f"获取收盘价失败 {symbol} @ {date}: {e}")
            return None

    def _get_tick_data(self, symbol: str, dt: datetime) -> Optional[Dict]:
        """获取Tick数据"""
        date_str = dt.strftime('%Y-%m-%d %H:%M:%S')
        tick_result = self.api_client.market.get_tick(code=symbol, date=date_str)

        if tick_result and symbol in tick_result:
            return tick_result[symbol]

        return None

    def _get_vkline_data(
        self,
        symbol: str,
        dt: datetime,
        frequency: str
    ) -> Optional[Dict]:
        """获取虚拟K线数据"""
        period = '1d' if frequency == 'daily' else '1m'
        date_str = dt.strftime('%Y-%m-%d %H:%M:%S')

        result = self.api_client.market.get_single_vkline(
            code=symbol,
            period=period,
```

```
            date=date_str
        )

        if result and symbol in result and len(result[symbol]) > 0:
            return result[symbol][0]

        return None
```

---

6. 交易撮合机制

6.1 撮合引擎设计理念

撮合引擎是框架的核心组件之一，负责将用户的订单转化为实际成交。设计遵循真实市场规则，确保回测结果的可靠性。

核心原则：

1. **真实性**：模拟真实市场的撮合逻辑
2. **保守性**：宁可拒绝订单，也不要过度乐观
3. **简洁性**：仅支持限价单和市价单，不支持复杂订单类型
4. **因果一致**：严格区分即时订单和历史订单的撮合逻辑

6.2 撮合规则详解

6.2.1 价格确定规则

市价单：

- 买入：使用ask1（卖一价），如ask1不存在则使用current_price
- 卖出：使用bid1（买一价），如bid1不存在则使用current_price

限价单：分为两种情况

1. 即时订单（当前bar提交的订单）：

- 买入：
  - 检查条件：limit_price >= ask1（或current_price）
  - 成交价格：ask1（或current_price），**而非用户限价**
  - 如不满足条件，订单保持OPEN状态，转为历史订单
- 卖出：
  - 检查条件：limit_price <= bid1（或current_price）
  - 成交价格：bid1（或current_price），**而非用户限价**
  - 如不满足条件，订单保持OPEN状态，转为历史订单

2. 历史订单（之前bar提交但未成交的订单）：

- 买入：
  - 检查条件：current_price <= limit_price（当前价格低于限价）
  - 成交价格：**用户限价**，而非ask1或current_price
  - 原因：避免未来函数，不能使用跨时间点的ask1
- 卖出：
  - 检查条件：current_price >= limit_price（当前价格高于限价）
  - 成交价格：**用户限价**，而非bid1或current_price
  - 原因：避免未来函数，不能使用跨时间点的bid1

为什么要区分即时订单和历史订单？

这是为了保证回测的因果一致性：

- 即时订单：知道当前时刻的市场状态（ask1/bid1），可以用市场价成交
- 历史订单：跨越了时间点，不能使用未来的市场状态，只能用限价成交

举例说明：

```
10:00 提交限价买单，限价10.50，此时ask1=10.55，无法成交
10:01 检查历史订单，当前价格10.48 <= 10.50，可以成交
      成交价格：10.50（限价），而不是10.01的ask1
      原因：10:00时不知道10:01的ask1是多少，只能用限价成交
```

6.2.2 滑点处理

滑点仅适用于**成交价格确定后**，计算公式：

```
最终成交价 = 撮合价格 ± 滑点
- 买入：最终成交价 = 撮合价格 + 滑点
- 卖出：最终成交价 = 撮合价格 - 滑点
- 滑点 = 撮合价格 × 滑点率
```

6.2.3 涨跌停检查

检查时机：在确定撮合价格后、应用滑点前

检查规则：

1. **当前价格等于涨跌停价时拒绝**：
    - 如果 current_price == high_limit，拒绝买入订单
    - 如果 current_price == low_limit，拒绝卖出订单
2. **加滑点后超出涨跌停范围时拒绝**：
    - 如果 最终买入价 > high_limit，拒绝
    - 如果 最终卖出价 < low_limit，拒绝

6.2.4 订单生命周期管理

订单状态转换：

```
提交订单
   ↓
OPEN（即时订单）
   ↓
   ├─→ 当前bar可成交 → FILLED
   │
   └─→ 当前bar不可成交 → OPEN（转为历史订单）
         ↓
         ├─→ 后续bar检查可成交 → FILLED
         │
         └─→ 当日收盘仍未成交 → EXPIRED（日频）或保持OPEN（分钟/tick频）
```

日频订单过期策略：

- 日频：当日未成交的订单在收盘时自动过期（EXPIRED）

- 分钟频/Tick频：订单在当日内一直有效，直到成交或收盘

6.2.5 T+1/T+0规则

股票：T+1交易

- 当日买入的股票，当日不可卖出
- Position对象中：today_open_amount表示当日买入数量
- 可卖出数量：available_amount（不包含today_open_amount）

ETF：T+0交易

- 当日买入可当日卖出
- 判断方法：代码以'5'开头视为ETF

日终处理：

- 每日结算时，将today_open_amount加入available_amount
- 重置today_open_amount为0

6.3 MatchingEngine（撮合引擎）完整实现

```python
# trading/matching_engine.py

from datetime import datetime
from typing import Optional, Dict, List
from .order import Order, OrderStatus, OrderSide, OrderType
from .commission import CommissionCalculator
from .slippage import SlippageModel
from ..core.context import Context

class MatchingEngine:
    """
    撮合引擎

    负责将订单转化为实际成交。核心职责：
    1. 区分即时订单和历史订单
    2. 获取价格数据
    3. 检查各种条件（停牌、涨跌停、资金等）
    4. 确定成交价格（即时订单用市场价，历史订单用限价）
    5. 应用滑点和手续费
    6. 更新账户和持仓

    撮合规则详见设计文档第6.2节。
    """

    def __init__(self, context: Context, config: dict):
        """
        Args:
            context: 全局上下文
            config: 撮合配置
        """
        self.context = context
        self.config = config

        # 手续费计算器
        self.commission_calc = CommissionCalculator(config.get('commission', {}))

        # 滑点模型
        self.slippage_model = SlippageModel(config.get('slippage', {}))

    def match_orders(self, dt: datetime):
        """
        撮合当前所有未成交订单

        处理流程：
```

```
        1. 先处理即时订单（当前bar提交的）
        2. 再处理历史订单（之前bar提交未成交的）

        Args:
            dt: 当前时间
        """
        order_manager = self.context.order_manager

        # 获取所有OPEN状态的订单
        open_orders = order_manager.get_open_orders()

        # 分类：即时订单和历史订单
        immediate_orders = [o for o in open_orders if o.is_immediate]
        historical_orders = [o for o in open_orders if not o.is_immediate]

        # 先处理即时订单
        for order in immediate_orders:
            self._try_match_immediate_order(order, dt)

        # 再处理历史订单
        for order in historical_orders:
            self._try_match_historical_order(order, dt)

    def _try_match_immediate_order(self, order: Order, dt: datetime):
        """
        尝试撮合即时订单

        即时订单：当前bar提交的订单，可以使用当前时刻的ask1/bid1

        Args:
            order: 订单对象
            dt: 当前时间
        """
        # 步骤1：获取完整价格数据
        snapshot = self.context.data_provider.get_current_snapshot(
            order.symbol,
            dt,
            self.context.frequency
        )

        if snapshot is None:
            self.context.logger.warning(
                f"无法获取 {order.symbol} 价格数据，跳过撮合"
            )
            # 转为历史订单，下次再试
            order.mark_as_historical()
            return

        # 步骤2：检查停牌
        if snapshot.get('is_suspended', False):
            order.reject("标的停牌")
            self.context.logger.info(
                f"订单 {order.id} 拒绝：{order.symbol} 停牌"
            )
            return

        # 步骤3：检查涨跌停（当前价格等于涨跌停价时拒绝）
        if not self._check_limit_price_equal(order, snapshot):
            order.reject("当前价格触及涨跌停")
            self.context.logger.info(
                f"订单 {order.id} 拒绝：{order.symbol} 当前价格触及涨跌停"
            )
            return

        # 步骤4：确定撮合价格（即时订单用市场价）
        match_price = self._determine_immediate_match_price(order, snapshot)
        if match_price is None:
            # 限价单无法成交，转为历史订单
            order.mark_as_historical()
            self.context.logger.debug(
                f"订单 {order.id} 暂时无法成交，转为历史订单"
            )
            return
```

```
        # 步骤5-9：执行通用撮合流程
        self._execute_match(order, match_price, snapshot, dt)

    def _try_match_historical_order(self, order: Order, dt: datetime):
        """
        尝试撮合历史订单

        历史订单：之前bar提交但未成交的订单，只能使用限价成交

        Args:
            order: 订单对象
            dt: 当前时间
        """
        # 市价单不应该成为历史订单
        if order.order_type == OrderType.MARKET:
            order.reject("市价单不应该成为历史订单")
            return

        # 步骤1：获取价格数据
        snapshot = self.context.data_provider.get_current_snapshot(
            order.symbol,
            dt,
            self.context.frequency
        )

        if snapshot is None:
            # 暂时无法获取数据，保持OPEN状态
            return

        # 步骤2：检查停牌
        if snapshot.get('is_suspended', False):
            # 停牌期间保持OPEN状态，不拒绝
            return

        # 步骤3：检查是否可以成交（历史订单用限价判断）
        current_price = snapshot.get('current_price')
        limit_price = order.limit_price

        can_match = False
        if order.side == OrderSide.BUY:
            # 买入：当前价格 <= 限价，可以成交
            if current_price <= limit_price:
                can_match = True
        else:
            # 卖出：当前价格 >= 限价，可以成交
            if current_price >= limit_price:
                can_match = True

        if not can_match:
            # 仍然无法成交，保持OPEN状态
            return

        # 步骤4：历史订单以限价成交（避免未来函数）
        match_price = limit_price

        self.context.logger.debug(
            f"历史订单 {order.id} 可以成交，以限价 {match_price:.2f} 成交"
        )

        # 步骤5-9：执行通用撮合流程
        self._execute_match(order, match_price, snapshot, dt)

    def _execute_match(
        self,
        order: Order,
        match_price: float,
        snapshot: Dict,
        dt: datetime
    ):
        """
        执行撮合的通用流程

        Args:
            order: 订单对象
```

```
            match_price: 撮合价格
            snapshot: 价格快照
            dt: 当前时间
        """
        # 步骤5：应用滑点
        slippage = self.slippage_model.calculate(order, match_price)
        if order.side == OrderSide.BUY:
            final_price = match_price + slippage
        else:
            final_price = match_price - slippage

        # 步骤6：检查加滑点后是否超出涨跌停范围
        if not self._check_limit_price_range(final_price, snapshot, order.side):
            order.reject("加滑点后超出涨跌停范围")
            self.context.logger.info(
                f"订单 {order.id} 拒绝：加滑点后超出涨跌停范围"
            )
            return

        # 步骤7：计算手续费
        commission = self.commission_calc.calculate(order, final_price)

        # 步骤8：检查资金/持仓是否足够
        if not self._check_sufficiency(order, final_price, commission):
            order.reject("资金/持仓不足")
            self.context.logger.warning(
                f"订单 {order.id} 拒绝：资金/持仓不足"
            )
            return

        # 步骤9：执行成交
        self._execute_order(order, final_price, commission, dt)

    def _determine_immediate_match_price(
        self,
        order: Order,
        snapshot: Dict
    ) -> Optional[float]:
        """
        确定即时订单的撮合价格

        即时订单使用市场价（ask1/bid1或current_price）

        Args:
            order: 订单对象
            snapshot: 价格快照

        Returns:
            撮合价格，如果无法成交返回None
        """
        current_price = snapshot.get('current_price')
        ask1 = snapshot.get('ask1')
        bid1 = snapshot.get('bid1')

        if order.order_type == OrderType.MARKET:
            # 市价单：直接使用档一价格
            if order.side == OrderSide.BUY:
                return ask1 if ask1 else current_price
            else:
                return bid1 if bid1 else current_price

        else:
            # 限价单：检查能否成交
            limit_price = order.limit_price

            if order.side == OrderSide.BUY:
                # 买入：限价 >= ask1（或current_price），才能成交
                reference_price = ask1 if ask1 else current_price
                if limit_price >= reference_price:
                    # 成交价为ask1（或current_price），而非限价
                    return reference_price
                else:
                    # 无法成交
                    return None
```

```
            else:
                # 卖出：限价 <= bid1（或current_price），才能成交
                reference_price = bid1 if bid1 else current_price
                if limit_price <= reference_price:
                    # 成交价为bid1（或current_price），而非限价
                    return reference_price
                else:
                    # 无法成交
                    return None

    def _check_limit_price_equal(
        self,
        order: Order,
        snapshot: Dict
    ) -> bool:
        """
        检查涨跌停（当前价格等于涨跌停价时拒绝）

        Args:
            order: 订单对象
            snapshot: 价格快照

        Returns:
            True表示通过检查，False表示拒绝
        """
        current_price = snapshot.get('current_price')
        high_limit = snapshot.get('high_limit')
        low_limit = snapshot.get('low_limit')

        if not high_limit or not low_limit:
            # 无涨跌停数据，放行
            return True

        # 使用0.01作为浮点数比较的容差
        epsilon = 0.01

        # 买入：当前价格等于涨停价，拒绝
        if order.side == OrderSide.BUY:
            if abs(current_price - high_limit) < epsilon:
                return False
        # 卖出：当前价格等于跌停价，拒绝
        else:
            if abs(current_price - low_limit) < epsilon:
                return False

        return True

    def _check_limit_price_range(
        self,
        price: float,
        snapshot: Dict,
        side: OrderSide
    ) -> bool:
        """
        检查价格是否超出涨跌停范围

        Args:
            price: 检查的价格（加滑点后的最终价格）
            snapshot: 价格快照
            side: 买卖方向

        Returns:
            True表示在范围内，False表示超出范围
        """
        high_limit = snapshot.get('high_limit')
        low_limit = snapshot.get('low_limit')

        if not high_limit or not low_limit:
            return True

        # 买入：不能超过涨停价
        if side == OrderSide.BUY:
            if price > high_limit:
                return False
```

```
            # 卖出：不能低于跌停价
            else:
                if price < low_limit:
                    return False

            return True

        def _check_sufficiency(
            self,
            order: Order,
            price: float,
            commission: float
        ) -> bool:
            """
            检查资金/持仓是否足够

            Args:
                order: 订单对象
                price: 成交价格
                commission: 手续费

            Returns:
                True表示足够，False表示不足
            """
            portfolio = self.context.portfolio

            if order.side == OrderSide.BUY:
                # 买入：检查现金
                total_cost = price * order.amount + commission
                return portfolio.cash >= total_cost
            else:
                # 卖出：检查持仓
                position = self.context.position_manager.get_position(order.symbol)
                if not position:
                    return False

                # 检查T+1规则
                if self._is_t_plus_1(order.symbol):
                    # 股票T+1，检查可用数量
                    return position.available_amount >= order.amount
                else:
                    # ETF T+0，检查总数量
                    return position.total_amount >= order.amount

        def _execute_order(
            self,
            order: Order,
            price: float,
            commission: float,
            dt: datetime
        ):
            """
            执行订单成交

            更新：
            1. 订单状态
            2. 账户现金
            3. 持仓信息

            Args:
                order: 订单对象
                price: 成交价格
                commission: 手续费
                dt: 成交时间
            """
            # 标记订单为已成交
            order.fill(price, commission, dt)

            # 更新账户和持仓
            portfolio = self.context.portfolio
            position_manager = self.context.position_manager

            if order.side == OrderSide.BUY:
                # 买入：减少现金，增加持仓
```

```
            total_cost = price * order.amount + commission
            portfolio.cash -= total_cost

            position_manager.increase_position(
                order.symbol,
                order.amount,
                price,
                commission,
                dt
            )

            self.context.logger.info(
                f"订单成交 - 买入 {order.symbol} {order.amount}股 "
                f"@{price:.2f} 手续费{commission:.2f} "
                f"({'即时' if order.is_immediate else '历史'}订单)"
            )
        else:
            # 卖出：增加现金，减少持仓
            total_value = price * order.amount - commission
            portfolio.cash += total_value

            pnl = position_manager.decrease_position(
                order.symbol,
                order.amount,
                price,
                commission,
                dt
            )

            self.context.logger.info(
                f"订单成交 - 卖出 {order.symbol} {order.amount}股 "
                f"@{price:.2f} 手续费{commission:.2f} 盈亏{pnl:.2f} "
                f"({'即时' if order.is_immediate else '历史'}订单)"
            )

        # 记录成交订单
        self.context.order_manager.add_filled_order(order)

    def settle(self):
        """
        每日收盘结算

        任务：
        1. 过期未成交订单（日频模式）
        2. 更新所有持仓的市值（使用收盘价）
        3. 更新账户总资产
        4. 记录账户历史
        5. 处理T+1（将今日买入加入可卖出）
        """
        self.context.logger.info("开始每日结算")

        # 1. 处理未成交订单（日频模式下过期）
        if self.context.frequency == 'daily':
            order_manager = self.context.order_manager
            open_orders = order_manager.get_open_orders()
            for order in open_orders:
                order.expire()
                self.context.logger.info(
                    f"订单 {order.id} 过期：{order.symbol} "
                    f"{'买入' if order.side == OrderSide.BUY else '卖出'} "
                    f"{order.amount}股"
                )

        # 2. 更新所有持仓市值
        position_manager = self.context.position_manager
        for position in position_manager.get_all_positions():
            close_price = self.context.data_provider.get_close_price(
                position.symbol,
                self.context.current_dt.strftime('%Y-%m-%d')
            )
            if close_price:
                position.update_price(close_price)

        # 3. 更新账户总值
```

```
        portfolio = self.context.portfolio
        total_value = portfolio.cash + sum(
            pos.market_value for pos in position_manager.get_all_positions()
        )
        portfolio.total_value = total_value

        # 4. 记录账户历史
        portfolio.record_history()

        # 5. 处理T+1
        for position in position_manager.get_all_positions():
            if self._is_t_plus_1(position.symbol):
                # 今日买入的明日可卖
                position.available_amount = position.total_amount
                position.today_open_amount = 0

        self.context.logger.info(f"结算完成 - 总资产: {total_value:.2f}")

    def _is_t_plus_1(self, symbol: str) -> bool:
        """
        判断是否T+1交易

        规则：
        - ETF代码通常以'5'开头，为T+0
        - 其他为T+1

        Args:
            symbol: 股票代码

        Returns:
            True表示T+1，False表示T+0
        """
        return not symbol.startswith('5')
```

6.4 SlippageModel（滑点模型）

```
# trading/slippage.py

from typing import Dict
from .order import Order

class SlippageModel:
    """
    滑点模型

    计算订单的滑点。目前仅支持固定滑点。
    """

    def __init__(self, config: Dict):
        """
        Args:
            config: 滑点配置
        """
        self.config = config
        self.type = config.get('type', 'fixed')  # 目前仅支持'fixed'
        self.rate = config.get('rate', 0.001)    # 固定滑点率（默认0.1%）

    def calculate(self, order: Order, price: float) -> float:
        """
        计算滑点

        Args:
            order: 订单对象
            price: 撮合价格

        Returns:
            滑点金额（绝对值）
        """
        if self.type == 'fixed':
            return self._fixed_slippage(price)
        else:
```

```
            return 0

    def _fixed_slippage(self, price: float) -> float:
        """
        固定滑点

        计算公式：滑点 = 价格 × 滑点率

        Args:
            price: 价格

        Returns:
            滑点金额
        """
        return price * self.rate
```

6.5 CommissionCalculator（手续费计算器）

```
# trading/commission.py

from typing import Dict
from .order import Order, OrderSide

class CommissionCalculator:
    """
    手续费计算器

    根据配置计算交易手续费（佣金+印花税）。
    """

    def __init__(self, config: Dict):
        """
        Args:
            config: 手续费配置
        """
        # 默认费率（A股标准）
        self.buy_commission = config.get('buy_commission', 0.0002)  # 买入佣金率
        self.sell_commission = config.get('sell_commission', 0.0002)  # 卖出佣金率
        self.buy_tax = config.get('buy_tax', 0.0)  # 买入印花税（A股为0）
        self.sell_tax = config.get('sell_tax', 0.001)  # 卖出印花税（A股为0.1%）
        self.min_commission = config.get('min_commission', 5.0)  # 最低佣金

    def calculate(self, order: Order, price: float) -> float:
        """
        计算手续费

        计算公式：
        - 佣金 = max(成交金额 × 佣金率, 最低佣金)
        - 印花税 = 成交金额 × 印花税率
        - 总手续费 = 佣金 + 印花税

        Args:
            order: 订单对象
            price: 成交价格

        Returns:
            总手续费
        """
        total_value = price * order.amount

        if order.side == OrderSide.BUY:
            commission = total_value * self.buy_commission
            tax = total_value * self.buy_tax
        else:
            commission = total_value * self.sell_commission
            tax = total_value * self.sell_tax

        # 佣金不足最低值时，按最低值收取
        commission = max(commission, self.min_commission)

        return commission + tax
```

7. 时间与事件调度系统

7.1 时间推进机制

时间推进是框架的核心机制之一。框架支持两种运行模式：

1. **回测模式（backtest）**：遍历历史交易日，逐日或逐分钟/tick推进
2. **模拟盘模式（simulation）**：使用系统时钟，实时推进

7.2 Scheduler（调度器）详细设计

Scheduler是事件调度的核心，负责：

1. 管理时间序列的推进
2. 在正确的时间点触发生命周期钩子
3. 协调撮合引擎执行
4. 支持日频多时间点执行

7.2.1 回测模式时间推进

```
日频回测流程（支持多时间点）：
for 交易日 in 交易日列表:
    09:25 → before_trading()
    for 时间点 in handle_bar_times:  # 例如 ["10:00:00", "14:55:00"]
        handle_bar()
        matching_engine.match_orders()  # 先撮合即时订单，再撮合历史订单
    15:01 → after_trading()
    15:30 → broker_settle()
    15:30 → matching_engine.settle()
    15:30 → benchmark_manager.update_daily()

分钟频回测流程：
for 交易日 in 交易日列表:
    09:25 → before_trading()
    for 分钟 in [09:30, 09:31, ..., 15:00]:
        handle_bar()
        matching_engine.match_orders()  # 每分钟检查历史订单
    15:01 → after_trading()
    15:30 → broker_settle()
    15:30 → matching_engine.settle()
    15:30 → benchmark_manager.update_daily()

Tick频回测流程：
for 交易日 in 交易日列表:
    09:25 → before_trading()
    for tick时间 in [09:30:00, 09:30:03, ..., 15:00:00]:
        handle_bar()
        matching_engine.match_orders()  # 每3秒检查历史订单
    15:01 → after_trading()
    15:30 → broker_settle()
    15:30 → matching_engine.settle()
    15:30 → benchmark_manager.update_daily()
```

7.2.2 Scheduler完整实现

```
# core/scheduler.py

from datetime import datetime, time, timedelta
```

```python
from typing import Optional, List
import time as time_module
from .context import Context
from .time_manager import TimeManager
from .lifecycle import LifecycleManager
from ..trading.matching_engine import MatchingEngine
from ..strategy.base import Strategy

class Scheduler:
    """
    事件调度器

    负责时间推进和事件调度。支持三种频率：
    - daily：支持多时间点执行
    - minute：每分钟一次
    - tick：每3秒一次（带精确时间校准）

    在回测模式下，遍历历史交易日推进时间。
    在模拟盘模式下，使用系统时钟实时推进。
    """

    # 定义交易时段（可通过配置覆盖）
    MORNING_START = time(9, 30)
    MORNING_END = time(11, 30)
    AFTERNOON_START = time(13, 0)
    AFTERNOON_END = time(15, 0)

    def __init__(
        self,
        context: Context,
        time_manager: TimeManager,
        matching_engine: MatchingEngine
    ):
        """
        Args:
            context: 全局上下文
            time_manager: 时间管理器
            matching_engine: 撮合引擎
        """
        self.context = context
        self.time_manager = time_manager
        self.matching_engine = matching_engine

        # 生命周期管理器
        self.lifecycle_manager = LifecycleManager(context)

        # 策略对象（由外部注册）
        self.strategy: Optional[Strategy] = None

        # 根据频率构建时间点列表（用于回测）
        if context.frequency == 'minute':
            self.time_points = self._build_minute_schedule()
        elif context.frequency == 'tick':
            self.time_points = self._build_tick_schedule()
        else:
            # 日频：从配置获取多个时间点
            self.time_points = self._get_handle_bar_times()

        # Tick频率的上次执行时间（用于模拟盘）
        self.last_tick_time: Optional[datetime] = None

    def _get_handle_bar_times(self) -> List[str]:
        """
        获取日频handle_bar执行时间点列表

        Returns:
            时间点列表，例如 ["10:00:00", "14:55:00"]
        """
        config_times = self.context.config.get('handle_bar_times')

        if config_times:
            # 配置了多个时间点
            if isinstance(config_times, list):
                return config_times
```

```
            else:
                # 单个时间点
                return [config_times]
        else:
            # 默认单个时间点14:55
            return [self.context.config.get('handle_bar_time', '14:55:00')]

    def register_strategy(self, strategy: Strategy):
        """
        注册策略对象

        Args:
            strategy: 用户策略实例
        """
        self.strategy = strategy
        self.lifecycle_manager.register_strategy(strategy)

    def run(self):
        """主运行循环入口"""
        if self.context.mode == 'backtest':
            self._run_backtest()
        else:
            self._run_simulation()

    def _run_backtest(self):
        """回测模式运行"""
        # 获取交易日列表
        trading_days = self.time_manager.get_trading_days(
            self.context.start_date,
            self.context.end_date
        )

        self.context.logger.info(f"交易日总数: {len(trading_days)}")

        # 调用初始化
        self.lifecycle_manager.call_initialize()

        # 遍历每个交易日
        for idx, current_date in enumerate(trading_days):
            if not self.context.is_running or self.context.is_paused:
                break

            self.context.logger.info(
                f"回测日期: {current_date} ({idx+1}/{len(trading_days)})"
            )

            # 根据频率执行
            if self.context.frequency == 'daily':
                self._run_daily_bar(current_date)
            elif self.context.frequency == 'minute':
                self._run_minute_bars(current_date)
            elif self.context.frequency == 'tick':
                self._run_tick_bars(current_date)

            # 更新可视化
            self._update_visualization()

        # 调用结束钩子
        self.lifecycle_manager.call_on_end()

    def _run_simulation(self):
        """模拟盘模式运行"""
        # 调用初始化
        self.lifecycle_manager.call_initialize()

        self.context.logger.info("进入模拟盘运行模式")

        while self.context.is_running:
            now = datetime.now()

            # 判断是否在交易日
            if not self.time_manager.is_trading_day(now):
                self.context.logger.debug(f"{now.date()} 非交易日，休眠")
                time_module.sleep(60)
```

```
                continue

            # 判断当前应该执行什么事件
            if self._should_run_before_trading(now):
                self.context.current_dt = now
                self.lifecycle_manager.call_before_trading()

            elif self._should_run_handle_bar(now):
                self.context.current_dt = now

                # 执行handle_bar
                self.lifecycle_manager.call_handle_bar()

                # 撮合（包括即时和历史订单）
                self.matching_engine.match_orders(now)

                # 更新可视化
                self._update_visualization()

            elif self._should_run_after_trading(now):
                self.context.current_dt = now
                self.lifecycle_manager.call_after_trading()

                # 更新可视化
                self._update_visualization()

            elif self._should_run_broker_settle(now):
                self.context.current_dt = now

                # broker_settle
                self.lifecycle_manager.call_broker_settle()

                # 日终结算
                self.matching_engine.settle()

                # 更新基准
                self.context.benchmark_manager.update_daily()

                self.context.logger.info("当日所有流程完成")

                # 更新可视化
                self._update_visualization()

            # 休眠到下一个事件点
            self._sleep_until_next_event()

    def _run_daily_bar(self, date: str):
        """执行单日回测（日频，支持多时间点）"""
        # 盘前（09:25）
        dt = datetime.strptime(f"{date} 09:25:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt
        self.lifecycle_manager.call_before_trading()

        # 盘中（多个时间点）
        for time_str in self.time_points:
            dt = datetime.strptime(f"{date} {time_str}", "%Y-%m-%d %H:%M:%S")
            self.context.current_dt = dt

            self.context.logger.debug(f"执行handle_bar @ {time_str}")

            # 调用handle_bar
            self.lifecycle_manager.call_handle_bar()

            # 撮合（先即时订单，再历史订单）
            self.matching_engine.match_orders(dt)

        # 盘后（15:01）
        dt = datetime.strptime(f"{date} 15:01:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt
        self.lifecycle_manager.call_after_trading()

        # broker_settle（15:30）
        broker_settle_time = self.context.config.get('broker_settle_time', '15:30:00')
        dt = datetime.strptime(f"{date} {broker_settle_time}", "%Y-%m-%d %H:%M:%S")
```

```
        self.context.current_dt = dt
        self.lifecycle_manager.call_broker_settle()

        # 日终结算
        self.matching_engine.settle()

        # 更新基准
        self.context.benchmark_manager.update_daily()

    def _run_minute_bars(self, date: str):
        """执行单日回测（分钟频）"""
        # 盘前
        dt = datetime.strptime(f"{date} 09:25:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt
        self.lifecycle_manager.call_before_trading()

        # 遍历分钟时间点
        for minute_str in self.time_points:
            dt = datetime.strptime(f"{date} {minute_str}", "%Y-%m-%d %H:%M:%S")
            self.context.current_dt = dt

            self.lifecycle_manager.call_handle_bar()
            # 每分钟都检查历史订单
            self.matching_engine.match_orders(dt)

            if self.context.is_paused:
                break

        # 盘后
        dt = datetime.strptime(f"{date} 15:01:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt
        self.lifecycle_manager.call_after_trading()

        # broker_settle
        broker_settle_time = self.context.config.get('broker_settle_time', '15:30:00')
        dt = datetime.strptime(f"{date} {broker_settle_time}", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt
        self.lifecycle_manager.call_broker_settle()

        # 日终结算
        self.matching_engine.settle()

        # 更新基准
        self.context.benchmark_manager.update_daily()

    def _run_tick_bars(self, date: str):
        """执行单日回测（Tick频，3秒一次）"""
        # 盘前
        dt = datetime.strptime(f"{date} 09:25:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt
        self.lifecycle_manager.call_before_trading()

        # 遍历Tick时间点
        for tick_str in self.time_points:
            dt = datetime.strptime(f"{date} {tick_str}", "%Y-%m-%d %H:%M:%S")
            self.context.current_dt = dt

            self.lifecycle_manager.call_handle_bar()
            # 每个tick都检查历史订单
            self.matching_engine.match_orders(dt)

            if self.context.is_paused:
                break

        # 盘后
        dt = datetime.strptime(f"{date} 15:01:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt
        self.lifecycle_manager.call_after_trading()

        # broker_settle
        broker_settle_time = self.context.config.get('broker_settle_time', '15:30:00')
        dt = datetime.strptime(f"{date} {broker_settle_time}", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt
        self.lifecycle_manager.call_broker_settle()
```

```
        # 日终结算
        self.matching_engine.settle()

        # 更新基准
        self.context.benchmark_manager.update_daily()

    def _build_minute_schedule(self) -> list:
        """构建分钟时间点列表"""
        schedule = []

        # 上午时段
        current = datetime.combine(datetime.today(), self.MORNING_START)
        end = datetime.combine(datetime.today(), self.MORNING_END)
        while current <= end:
            schedule.append(current.strftime("%H:%M:%S"))
            current += timedelta(minutes=1)

        # 下午时段
        current = datetime.combine(datetime.today(), self.AFTERNOON_START)
        end = datetime.combine(datetime.today(), self.AFTERNOON_END)
        while current <= end:
            schedule.append(current.strftime("%H:%M:%S"))
            current += timedelta(minutes=1)

        return schedule

    def _build_tick_schedule(self) -> list:
        """构建Tick时间点列表（3秒一次）"""
        schedule = []

        # 上午时段
        current = datetime.combine(datetime.today(), self.MORNING_START)
        end = datetime.combine(datetime.today(), self.MORNING_END)
        while current <= end:
            schedule.append(current.strftime("%H:%M:%S"))
            current += timedelta(seconds=3)

        # 下午时段
        current = datetime.combine(datetime.today(), self.AFTERNOON_START)
        end = datetime.combine(datetime.today(), self.AFTERNOON_END)
        while current <= end:
            schedule.append(current.strftime("%H:%M:%S"))
            current += timedelta(seconds=3)

        return schedule

    def _should_run_before_trading(self, now: datetime) -> bool:
        """判断是否应该执行before_trading"""
        return now.time() >= time(9, 25) and now.time() < time(9, 26)

    def _should_run_handle_bar(self, now: datetime) -> bool:
        """判断是否应该执行handle_bar"""
        current_time = now.time()

        # 判断是否在交易时段
        in_trading_hours = (
            (self.MORNING_START <= current_time <= self.MORNING_END) or
            (self.AFTERNOON_START <= current_time <= self.AFTERNOON_END)
        )

        if not in_trading_hours:
            return False

        # 日频：检查是否在配置的时间点列表中
        if self.context.frequency == 'daily':
            for time_str in self.time_points:
                target_time = datetime.strptime(time_str, '%H:%M:%S').time()
                if current_time >= target_time and current_time < time(target_time.hour, target_time.minute +
                    return True
            return False

        # 分钟频：每分钟整点执行
        elif self.context.frequency == 'minute':
```

```
            return now.second == 0

        # Tick频：每3秒执行（带时间校准）
        elif self.context.frequency == 'tick':
            return self._should_run_tick(now)

        return False

    def _should_run_tick(self, now: datetime) -> bool:
        """
        判断Tick频率是否应该执行（带时间校准）

        关键逻辑：
        1. 计算到下一个3秒整数倍的时间
        2. 如果当前时间已经到达或超过，立即执行
        3. 避免累积误差
        """
        # 计算当前时间是否为3秒的整数倍
        if now.second % 3 == 0:
            # 检查是否刚执行过（避免同一秒重复执行）
            if self.last_tick_time is None or \
               (now - self.last_tick_time).total_seconds() >= 3:
                self.last_tick_time = now
                return True

        # 检查是否超时（handle_bar执行时间超过3秒）
        if self.last_tick_time is not None:
            elapsed = (now - self.last_tick_time).total_seconds()
            if elapsed >= 3:
                # 立即执行，不等待下一个整数倍
                self.last_tick_time = now
                return True

        return False

    def _should_run_after_trading(self, now: datetime) -> bool:
        """判断是否应该执行after_trading"""
        return now.time() >= time(15, 1) and now.time() < time(15, 2)

    def _should_run_broker_settle(self, now: datetime) -> bool:
        """判断是否应该执行broker_settle"""
        broker_settle_time_str = self.context.config.get('broker_settle_time', '15:30:00')
        broker_settle_time = datetime.strptime(broker_settle_time_str, '%H:%M:%S').time()

        return (
            now.time() >= broker_settle_time and
            now.time() < time(broker_settle_time.hour, broker_settle_time.minute + 1)
        )

    def _sleep_until_next_event(self):
        """
        休眠到下一个事件点

        Tick频率的精确时间校准：
        1. 计算到下一个3秒整数倍的精确时间
        2. 计算需要休眠的秒数
        3. 如果计算结果为负数（已过期），则休眠1秒后重试
        """
        if self.context.frequency == 'tick':
            # Tick频：精确计算到下一个3秒整数倍的时间
            now = datetime.now()
            current_second = now.second

            # 计算到下一个3秒整数倍需要的秒数
            next_second = ((current_second // 3) + 1) * 3
            if next_second >= 60:
                next_second = 0
                sleep_seconds = 60 - current_second
            else:
                sleep_seconds = next_second - current_second

            # 减去当前的微秒，保证精确
            sleep_seconds -= now.microsecond / 1000000.0
```

```
            # 休眠
            if sleep_seconds > 0:
                time_module.sleep(sleep_seconds)
            else:
                # 如果计算结果<=0，说明已经错过时间点，休眠1秒后重试
                time_module.sleep(1)

        elif self.context.frequency == 'minute':
            # 分钟频：休眠到下一分钟
            now = datetime.now()
            sleep_seconds = 60 - now.second - now.microsecond / 1000000.0
            if sleep_seconds > 0:
                time_module.sleep(sleep_seconds)
            else:
                time_module.sleep(1)

        else:
            # 日频：休眠5秒后再检查
            time_module.sleep(5)

    def _update_visualization(self):
        """更新可视化数据"""
        if self.context.visualization_server:
            self.context.visualization_server.update_data()
```

7.3 TimeManager（时间管理器）

```
# core/time_manager.py

from datetime import datetime, time
from typing import List
from .context import Context

class TimeManager:
    """
    时间管理器

    提供时间相关的工具方法，依赖用户注入的数据提供者。
    """

    MORNING_START = time(9, 30)
    MORNING_END = time(11, 30)
    AFTERNOON_START = time(13, 0)
    AFTERNOON_END = time(15, 0)

    def __init__(self, context: Context):
        """
        Args:
            context: 全局上下文
        """
        self.context = context

    def is_trading_day(self, dt: datetime) -> bool:
        """
        判断是否为交易日

        Args:
            dt: 日期时间

        Returns:
            True表示是交易日
        """
        if self.context.data_provider is None:
            raise RuntimeError("未注册数据提供者")

        date_str = dt.strftime('%Y-%m-%d')

        # 获取当天的交易日历（只包含一天）
        calendar = self.context.data_provider.get_trading_calendar(
            date_str,
            date_str
```

```
        )

        return len(calendar) > 0

    def is_trading_time(self, dt: datetime) -> bool:
        """
        判断是否在交易时段

        Args:
            dt: 日期时间

        Returns:
            True表示在交易时段
        """
        if not self.is_trading_day(dt):
            return False

        current_time = dt.time()
        return (
            (self.MORNING_START <= current_time <= self.MORNING_END) or
            (self.AFTERNOON_START <= current_time <= self.AFTERNOON_END)
        )

    def get_trading_days(self, start: str, end: str) -> List[str]:
        """
        获取交易日列表

        Args:
            start: 开始日期
            end: 结束日期

        Returns:
            交易日列表
        """
        if self.context.data_provider is None:
            raise RuntimeError("未注册数据提供者")

        return self.context.data_provider.get_trading_calendar(start, end)
```

8. 策略生命周期详解

8.1 生命周期钩子概述

QTrader框架提供了丰富的生命周期钩子，让用户可以在不同时间点执行策略逻辑。

完整生命周期钩子列表：

钩子名称	调用时机	频率	是否必须	说明
initialize	策略启动时	一次	是	初始化策略参数
before_trading	每日开盘前	每日一次	否	盘前准备工作
handle_bar	盘中	根据频率/配置	否	核心策略逻辑
after_trading	每日收盘后	每日一次	否	盘后处理
broker_settle	日终结算	每日一次	否	结算与对账
on_end	策略结束时	一次	否	清理工作

8.2 各钩子详细说明

8.2.1 initialize(context)

调用时机：策略启动时，在任何其他钩子之前调用一次

主要用途：

- 初始化策略参数
- 设置全局变量
- 准备数据结构

参数：

- context: 全局上下文对象

注意事项：

- 这是唯一的必须实现的钩子
- 此时账户已创建，但尚未开始交易
- 可以使用context.set()保存自定义数据

示例：

```
def initialize(context):
    """策略初始化"""
    # 设置策略参数
    context.set('ma_short', 5)
    context.set('ma_long', 20)

    # 初始化数据结构
    context.set('price_history', {})

    # 设置股票池
    context.set('universe', ['000001', '600519', '000858'])

    context.logger.info("策略初始化完成")
```

8.2.2 before_trading(context, data)

调用时机：每日开盘前（默认09:25）

主要用途：

- 更新股票池
- 读取当日要交易的股票列表
- 准备当日所需数据

参数：

- context: 全局上下文对象
- data: 数据代理对象（用户可通过context.data_provider获取数据）

调用时间：

- 回测：每日09:25
- 模拟盘：每日09:25（实时触发）

注意事项：

- 此时市场尚未开盘，无法获取当日实时价格

- 可以使用前一日的收盘价
- 用户需要自行通过data_provider获取历史数据

示例：

```
def before_trading(context, data):
    """盘前准备"""
    # 更新股票池
    universe = context.get('universe')

    # 打印当日将要关注的股票
    context.logger.info(f"今日关注股票: {universe}")

    # 准备当日数据（用户自行实现数据获取）
    context.set('today_signals', [])
```

8.2.3 handle_bar(context, data)

调用时机：盘中执行，频率和时间点取决于配置

主要用途：

- 核心策略逻辑
- 生成交易信号
- 提交订单

参数：

- context: 全局上下文对象
- data: 数据代理对象

调用时间（可配置）：

频率	调用时机	可配置	配置项
daily	配置的时间点列表	是	handle_bar_times
minute	每分钟	否	-
tick	每3秒	否	-

配置示例：

```
# configs/backtest.yaml

# 单时间点（默认）
handle_bar_time: "14:55:00"

# 或者多时间点
handle_bar_times:
  - "10:00:00"
  - "14:55:00"
```

注意事项：

- 这是策略的核心逻辑所在
- 可以调用order_manager提交订单
- 即时订单会立即尝试撮合，历史订单会在每次handle_bar时检查

- 用户需要自行通过data_provider获取历史数据用于因子计算

示例：

```
def handle_bar(context, data):
    """盘中策略执行"""
    symbol = '000001'

    # 用户自行获取历史数据（框架不负责）
    # 这里仅作示例，实际需要用户实现

    # 获取当前价格（框架提供，用于撮合）
    snapshot = context.data_provider.get_current_snapshot(
        symbol,
        context.current_dt,
        context.frequency
    )

    if snapshot is None:
        return

    current_price = snapshot['current_price']

    # 用户的策略逻辑...
    # 例如：基于某些条件提交订单

    # 获取当前持仓
    position = context.position_manager.get_position(symbol)

    # 交易逻辑
    if position is None or position.total_amount == 0:
        # 买入
        cash = context.portfolio.cash
        amount = int(cash * 0.5 / current_price / 100) * 100

        if amount > 0:
            context.order_manager.submit_market_order(symbol, amount)
            context.logger.info(f"买入信号: {symbol} {amount}股")
```

8.2.4 after_trading(context, data)

调用时机：每日收盘后（默认15:01）

主要用途：

- 检查当日成交情况
- 记录当日统计信息
- 准备明日数据

参数：

- context: 全局上下文对象
- data: 数据代理对象

调用时间：

- 回测：每日15:01
- 模拟盘：每日15:01（实时触发）

注意事项：

- 此时市场已收盘
- 可以获取当日收盘价

- 还未进行日终结算

示例：

```
def after_trading(context, data):
    """盘后处理"""
    # 检查当日成交情况
    filled_orders = context.order_manager.get_filled_orders_today()

    context.logger.info(f"今日成交订单数: {len(filled_orders)}")

    # 记录当日收益
    returns = context.portfolio.returns
    context.logger.info(f"当前总收益率: {returns:.2%}")
```

8.2.5 broker_settle(context)

调用时机：日终结算（默认15:30，可配置）

主要用途：

- 对账：对比模拟盘与实盘的差异
- 修正：调整模拟盘的持仓/资金以匹配实盘
- 准备：为明日交易准备数据

参数：

- context: 全局上下文对象

调用时间（可配置）：

- 默认：15:30
- 配置项：broker_settle_time

配置示例：

```
# configs/simulation.yaml
broker_settle_time: "16:00:00"  # 改为16:00执行
```

使用场景：

1. **模拟盘与实盘对账**：

```
    def broker_settle(context):
        """日终结算与对账"""
        # 假设用户有实盘账户API
        real_account = get_real_account_info()

        # 对比现金
        if abs(context.portfolio.cash - real_account['cash']) > 100:
            context.logger.warning(
                f"现金差异: 模拟盘{context.portfolio.cash}, "
                f"实盘{real_account['cash']}"
            )

            # 修正模拟盘
            context.portfolio.cash = real_account['cash']

        # 对比持仓
        for symbol, real_pos in real_account['positions'].items():
```

```
            sim_pos = context.position_manager.get_position(symbol)

            if sim_pos is None or sim_pos.total_amount != real_pos['amount']:
                context.logger.warning(
                    f"持仓差异: {symbol} 模拟盘{sim_pos.total_amount if sim_pos else 0}, "
                    f"实盘{real_pos['amount']}"
                )

                # 修正模拟盘持仓（用户暴露的接口）
                # context.position_manager.adjust_position(...)
```

2. **收集明日数据**：

```
    def broker_settle(context):
        """准备明日数据"""
        # 预取明日可能需要的数据
        universe = context.get('universe')

        # 用户可以在这里提前获取数据并缓存
        # 这样明日handle_bar时可以快速访问
```

注意事项：

- 此钩子在matching_engine.settle()之后执行
- 此时持仓的available_amount已更新（T+1处理）
- 可以直接修改context.portfolio和position的属性

8.2.6 on_end(context)

调用时机：策略结束时（回测完成或模拟盘停止）

主要用途：

- 清理资源
- 保存最终结果
- 打印统计信息

参数：

- context: 全局上下文对象

示例：

```
def on_end(context):
    """策略结束"""
    # 打印最终收益
    final_returns = context.portfolio.returns
    context.logger.info(f"策略最终收益率: {final_returns:.2%}")

    # 打印基准收益
    benchmark_returns = context.benchmark_manager.get_current_returns()
    context.logger.info(f"基准最终收益率: {benchmark_returns:.2%}")

    # 保存自定义数据
    import json
    with open('my_data.json', 'w') as f:
        json.dump(context.user_data, f)
```

8.3 LifecycleManager（生命周期管理器）

```python
# core/lifecycle.py

from typing import Optional
from .context import Context
from ..strategy.base import Strategy

class LifecycleManager:
    """
    生命周期管理器

    负责调用策略的各个生命周期钩子。
    """

    def __init__(self, context: Context):
        """
        Args:
            context: 全局上下文
        """
        self.context = context
        self.strategy: Optional[Strategy] = None

    def register_strategy(self, strategy: Strategy):
        """
        注册策略对象

        Args:
            strategy: 用户策略实例
        """
        self.strategy = strategy

    def call_initialize(self):
        """调用initialize钩子"""
        if hasattr(self.strategy, 'initialize'):
            self.context.logger.info("调用 initialize()")
            self.strategy.initialize(self.context)

    def call_before_trading(self):
        """调用before_trading钩子"""
        if hasattr(self.strategy, 'before_trading'):
            # data参数供用户通过context.data_provider获取数据
            self.strategy.before_trading(self.context, None)

    def call_handle_bar(self):
        """调用handle_bar钩子"""
        if hasattr(self.strategy, 'handle_bar'):
            # data参数供用户通过context.data_provider获取数据
            self.strategy.handle_bar(self.context, None)

    def call_after_trading(self):
        """调用after_trading钩子"""
        if hasattr(self.strategy, 'after_trading'):
            # data参数供用户通过context.data_provider获取数据
            self.strategy.after_trading(self.context, None)

    def call_broker_settle(self):
        """调用broker_settle钩子"""
        if hasattr(self.strategy, 'broker_settle'):
            self.strategy.broker_settle(self.context)

    def call_on_end(self):
        """调用on_end钩子"""
        if hasattr(self.strategy, 'on_end'):
            self.context.logger.info("调用 on_end()")
            self.strategy.on_end(self.context)
```

9. 基准管理系统

- **职责**:

    - 在策略初始化时，通过 `data_provider.get_history_bars` **一次性获取**基准标的在回测期内的**全部日线收盘价**并缓存于内存。
    - 在每日结算时，根据当天的日期，从内存中快速查找对应的基准收盘价，并计算当天的基准净值和收益率。
- **灵活性**: 用户可在配置文件中指定**任何**可以通过 `DataProvider` 获取到数据的标的作为基准。

9.1 BenchmarkManager

```python
# benchmark/benchmark_manager.py

from typing import List, Dict, Optional
from datetime import datetime
from ..core.context import Context

class BenchmarkManager:
    """
    基准管理器

    负责跟踪基准标的的表现，用于对比策略收益。

    核心功能：
    1. 每日收盘获取基准收盘价
    2. 计算基准收益率
    3. 提供基准对比数据
    """

    def __init__(self, context: Context, config: Dict):
        """
        Args:
            context: 全局上下文
            config: 基准配置
        """
        self.context = context
        self.config = config
        self.benchmark_symbol: Optional[str] = None

        # 基准历史数据
        self.benchmark_history: List[Dict] = []

        # 初始值
        self.initial_value: Optional[float] = None

    def initialize(self, symbol: str):
        """
        初始化基准

        Args:
            symbol: 基准标的代码（如 '000300' 表示沪深300）
        """
        self.benchmark_symbol = symbol

        # 获取初始值（start_date的收盘价）
        start_date = self.context.start_date
        self.initial_value = self.context.data_provider.get_close_price(
            symbol,
            start_date
        )

        if self.initial_value is None:
            self.context.logger.warning(
                f"无法获取基准 {symbol} 在 {start_date} 的价格，"
                f"使用默认值1000"
            )
            self.initial_value = 1000.0

        self.context.logger.info(
            f"基准初始化完成: {symbol}, 初始价格: {self.initial_value:.2f}"
        )

    def update_daily(self):
        """
        每日收盘更新基准数据
```

```
        在matching_engine.settle()之后调用
        """
        if self.benchmark_symbol is None:
            return

        current_date = self.context.current_dt.strftime('%Y-%m-%d')

        # 获取收盘价
        close_price = self.context.data_provider.get_close_price(
            self.benchmark_symbol,
            current_date
        )

        if close_price is None:
            self.context.logger.warning(
                f"无法获取基准 {self.benchmark_symbol} "
                f"在 {current_date} 的收盘价"
            )
            return

        # 计算收益率
        returns = (close_price - self.initial_value) / self.initial_value

        # 计算基准价值（假设初始投资与策略相同）
        value = self.context.portfolio.initial_cash * (1 + returns)

        # 记录历史
        self.benchmark_history.append({
            'date': current_date,
            'close_price': close_price,
            'returns': returns,
            'value': value,
        })

        self.context.logger.debug(
            f"基准更新: {current_date}, "
            f"收盘价: {close_price:.2f}, "
            f"收益率: {returns:.2%}"
        )

    def get_current_returns(self) -> float:
        """获取当前基准收益率"""
        if len(self.benchmark_history) == 0:
            return 0.0

        return self.benchmark_history[-1]['returns']

    def get_current_value(self) -> float:
        """获取当前基准价值"""
        if len(self.benchmark_history) == 0:
            return self.context.portfolio.initial_cash

        return self.benchmark_history[-1]['value']

    def get_benchmark_data(self) -> List[Dict]:
        """获取完整基准历史数据"""
        return self.benchmark_history
```

10. 账户与持仓管理

10.1 OrderManager（订单管理器）

- 管理订单生命周期：`OPEN` -> `FILLED` / `REJECTED` / `EXPIRED`。
- 提供下单接口给策略：`submit_market_order`，`submit_limit_order`。

```python
# trading/order_manager.py

from typing import Dict, List, Optional
from datetime import datetime
from .order import Order, OrderType, OrderSide, OrderStatus
from ..core.context import Context

class OrderManager:
    """
    订单管理器

    负责订单的创建、查询、管理。

    关键功能：
    - 维护当日订单列表（包括即时和历史订单）
    - 维护历史成交订单
    - 提供订单查询接口
    """

    def __init__(self, context: Context):
        """
        Args:
            context: 全局上下文
        """
        self.context = context

        # 当日订单列表（每日结算时清空）
        self.orders: Dict[str, Order] = {}

        # 历史成交订单（持久化）
        self.filled_orders: List[Order] = []

    def submit_market_order(
        self,
        symbol: str,
        amount: int
    ) -> Optional[str]:
        """
        提交市价单

        Args:
            symbol: 股票代码
            amount: 数量（正数买入，负数卖出）

        Returns:
            订单ID，失败返回None
        """
        if amount == 0:
            self.context.logger.warning("下单失败：数量为0")
            return None

        side = OrderSide.BUY if amount > 0 else OrderSide.SELL
        amount = abs(amount)

        order = Order(
            symbol=symbol,
            amount=amount,
            side=side,
            order_type=OrderType.MARKET
        )
        order.created_time = self.context.current_dt
        order.created_bar_time = self.context.current_dt
        order.is_immediate = True  # 标记为即时订单

        self.orders[order.id] = order

        self.context.logger.info(
            f"提交市价单: {symbol} {'买入' if side == OrderSide.BUY else '卖出'} "
            f"{amount}股"
        )

        return order.id
```

```python
    def submit_limit_order(
        self,
        symbol: str,
        amount: int,
        price: float
    ) -> Optional[str]:
        """
        提交限价单

        Args:
            symbol: 股票代码
            amount: 数量（正数买入，负数卖出）
            price: 限价

        Returns:
            订单ID，失败返回None
        """
        if amount == 0:
            self.context.logger.warning("下单失败：数量为0")
            return None

        side = OrderSide.BUY if amount > 0 else OrderSide.SELL
        amount = abs(amount)

        order = Order(
            symbol=symbol,
            amount=amount,
            side=side,
            order_type=OrderType.LIMIT,
            limit_price=price
        )
        order.created_time = self.context.current_dt
        order.created_bar_time = self.context.current_dt
        order.is_immediate = True  # 标记为即时订单

        self.orders[order.id] = order

        self.context.logger.info(
            f"提交限价单: {symbol} {'买入' if side == OrderSide.BUY else '卖出'} "
            f"{amount}股 @{price:.2f}"
        )

        return order.id

    def get_open_orders(self) -> List[Order]:
        """
        获取所有未成交订单

        Returns:
            未成交订单列表
        """
        return [
            order for order in self.orders.values()
            if order.status == OrderStatus.OPEN
        ]

    def get_filled_orders_today(self) -> List[Order]:
        """
        获取当日已成交订单

        Returns:
            当日已成交订单列表
        """
        return [
            order for order in self.orders.values()
            if order.status == OrderStatus.FILLED
        ]

    def get_filled_orders(self) -> List[Order]:
        """
        获取所有历史成交订单

        Returns:
            历史成交订单列表
```

```
        """
        return self.filled_orders

    def add_filled_order(self, order: Order):
        """
        添加已成交订单到历史记录

        Args:
            order: 已成交的订单对象
        """
        self.filled_orders.append(order)

    def get_all_orders(self) -> List[Order]:
        """
        获取所有订单（包括当日和历史）

        Returns:
            所有订单列表
        """
        return list(self.orders.values()) + self.filled_orders

    def clear_today_orders(self):
        """
        清空当日订单（在日终结算时调用）

        注意：只清空当日订单列表，已成交的订单已移至历史记录
        """
        self.orders.clear()

    def restore_orders(self, orders: List[Order]):
        """
        恢复订单（用于状态恢复）

        Args:
            orders: 订单列表
        """
        self.filled_orders = orders
```

10.2 PositionManager（持仓管理器）

- 管理所有持仓，精确处理 T+1 规则。
- `Position` 对象中 `total_amount`，`available_amount`，`today_open_amount` 三个属性协同工作，确保可卖数量的准确性。
- **新增接口**：`adjust_position`，允许用户在 `broker_settle` 中手动修正持仓，以匹配外部系统状态。

```
# trading/position_manager.py

from typing import Dict, List, Optional
from datetime import datetime
from .position import Position
from ..core.context import Context

class PositionManager:
    """
    持仓管理器

    负责持仓的创建、更新、查询。
    """

    def __init__(self, context: Context):
        """
        Args:
            context: 全局上下文
        """
        self.context = context

        # 持仓字典，key为股票代码
        self.positions: Dict[str, Position] = {}

    def get_position(self, symbol: str) -> Optional[Position]:
```

```
        """
        获取指定股票的持仓

        Args:
            symbol: 股票代码

        Returns:
            持仓对象，不存在返回None
        """
        return self.positions.get(symbol)

    def get_all_positions(self) -> List[Position]:
        """
        获取所有持仓

        Returns:
            持仓列表
        """
        return list(self.positions.values())

    def increase_position(
        self,
        symbol: str,
        amount: int,
        price: float,
        commission: float,
        dt: datetime
    ):
        """
        增加持仓（买入）

        Args:
            symbol: 股票代码
            amount: 买入数量
            price: 买入价格
            commission: 手续费
            dt: 交易时间
        """
        if symbol in self.positions:
            # 已有持仓，加仓
            position = self.positions[symbol]

            # 更新平均成本
            total_cost = (
                position.avg_cost * position.total_amount +
                price * amount + commission
            )
            position.total_amount += amount
            position.avg_cost = total_cost / position.total_amount

            # 更新今日开仓数量（T+1处理）
            position.today_open_amount += amount

        else:
            # 新建持仓
            avg_cost = (price * amount + commission) / amount
            position = Position(
                symbol=symbol,
                amount=amount,
                avg_cost=avg_cost,
                current_dt=dt
            )
            position.today_open_amount = amount

            self.positions[symbol] = position

        position.last_update_time = dt

    def decrease_position(
        self,
        symbol: str,
        amount: int,
        price: float,
        commission: float,
```

```
        dt: datetime
    ) -> float:
        """
        减少持仓（卖出）

        Args:
            symbol: 股票代码
            amount: 卖出数量
            price: 卖出价格
            commission: 手续费
            dt: 交易时间

        Returns:
            本次交易盈亏
        """
        if symbol not in self.positions:
            self.context.logger.error(f"卖出失败：无持仓 {symbol}")
            return 0.0

        position = self.positions[symbol]

        # 计算盈亏
        pnl = (price - position.avg_cost) * amount - commission

        # 减少持仓数量
        position.total_amount -= amount
        position.available_amount -= amount

        # 如果全部卖出，删除持仓
        if position.total_amount <= 0:
            del self.positions[symbol]
        else:
            position.last_update_time = dt

        return pnl

    def adjust_position(
        self,
        symbol: str,
        amount: int,
        avg_cost: float
    ):
        """
        调整持仓（用于broker_settle中修正持仓）

        这是暴露给用户的接口，允许在broker_settle中修正持仓。

        Args:
            symbol: 股票代码
            amount: 目标持仓数量
            avg_cost: 平均成本
        """
        if amount <= 0:
            # 删除持仓
            if symbol in self.positions:
                del self.positions[symbol]
        else:
            if symbol in self.positions:
                # 调整现有持仓
                position = self.positions[symbol]
                position.total_amount = amount
                position.available_amount = amount
                position.avg_cost = avg_cost
                position.today_open_amount = 0
            else:
                # 创建新持仓
                position = Position(
                    symbol=symbol,
                    amount=amount,
                    avg_cost=avg_cost,
                    current_dt=self.context.current_dt
                )
                position.available_amount = amount
                position.today_open_amount = 0
```

```
                self.positions[symbol] = position

        self.context.logger.info(
            f"持仓调整: {symbol} 数量={amount} 成本={avg_cost:.2f}"
        )

    def restore_positions(self, positions: List[Position]):
        """
        恢复持仓（用于状态恢复）

        Args:
            positions: 持仓列表
        """
        self.positions = {pos.symbol: pos for pos in positions}
```

11. 状态持久化与恢复

- **目标**: 仅支持 `simulation` 模式的暂停与恢复。
- **机制**: 每日收盘结算后，或接收到 `Ctrl+C` 等暂停指令时，框架自动将 `Context` 对象通过 `pickle` 序列化到磁盘。
- **限制**: 策略中不能在 `context.user_data` 中存储无法被 `pickle` 序列化的对象。

11.1 StateSerializer（状态序列化器）

```python
# utils/serializer.py

import pickle
import os
from datetime import datetime
from typing import Optional
from ..core.context import Context

class StateSerializer:
    """
    状态序列化器

    负责保存和加载策略运行状态。

    保存内容：
    - Context基本信息
    - Portfolio账户信息
    - Positions持仓信息
    - Orders订单信息
    - BenchmarkManager基准数据
    - user_data用户自定义数据
    """

    def __init__(self, context: Context, save_dir: str = '.states'):
        """
        Args:
            context: 全局上下文
            save_dir: 状态保存目录
        """
        self.context = context
        self.save_dir = save_dir
        os.makedirs(save_dir, exist_ok=True)

    def save(self, tag: Optional[str] = None):
        """
        保存当前状态

        Args:
            tag: 标签（默认使用当前日期）
        """
        if tag is None:
            tag = self.context.current_dt.strftime('%Y%m%d')
```

```
        file_path = os.path.join(
            self.save_dir,
            f"{self.context.strategy_name}_{tag}.pkl"
        )

        # 收集状态
        state = {
            'context': {
                'mode': self.context.mode,
                'strategy_name': self.context.strategy_name,
                'start_date': self.context.start_date,
                'end_date': self.context.end_date,
                'current_dt': self.context.current_dt,
                'frequency': self.context.frequency,
                'config': self.context.config,
            },
            'portfolio': self.context.portfolio,
            'positions': self.context.position_manager.get_all_positions(),
            'orders': self.context.order_manager.get_all_orders(),
            'benchmark_history': self.context.benchmark_manager.benchmark_history,
            'user_data': self.context.user_data,
            'timestamp': datetime.now().isoformat()
        }

        # 序列化
        with open(file_path, 'wb') as f:
            pickle.dump(state, f)

        self.context.logger.info(f"状态已保存到 {file_path}")

    def load(self, file_path: str):
        """
        加载状态

        Args:
            file_path: 状态文件路径
        """
        with open(file_path, 'rb') as f:
            state = pickle.load(f)

        # 恢复Context基本信息
        context_data = state['context']
        self.context.mode = context_data['mode']
        self.context.strategy_name = context_data['strategy_name']
        self.context.start_date = context_data['start_date']
        self.context.end_date = context_data['end_date']
        self.context.current_dt = context_data['current_dt']
        self.context.frequency = context_data['frequency']
        self.context.config = context_data['config']

        # 恢复Portfolio
        self.context.portfolio = state['portfolio']

        # 恢复Positions
        self.context.position_manager.restore_positions(state['positions'])

        # 恢复Orders
        self.context.order_manager.restore_orders(state['orders'])

        # 恢复Benchmark
        self.context.benchmark_manager.benchmark_history = state['benchmark_history']

        # 恢复user_data
        self.context.user_data = state['user_data']

        self.context.logger.info(f"状态已从 {file_path} 加载")
        self.context.logger.info(f"保存时间: {state['timestamp']}")
```

12. 实时可视化系统

12.1 VisualizationServer

```python
# visualization/server.py

from flask import Flask, render_template, jsonify
from flask_socketio import SocketIO
import threading
from typing import Dict, Any, Optional
from ..core.context import Context

class VisualizationServer:
    """
    可视化服务器

    提供Web界面实时展示回测/模拟盘运行状态。

    功能：
    - 实时监控面板
    - 权益曲线图
    - 持仓详情表
    - 订单记录表
    - 账户历史表
    """

    def __init__(self, context: Context, config: Dict):
        """
        Args:
            context: 全局上下文
            config: 可视化配置
        """
        self.context = context
        self.config = config

        # Flask应用
        self.app = Flask(
            __name__,
            template_folder='templates',
            static_folder='static'
        )

        # SocketIO
        self.socketio = SocketIO(self.app, cors_allowed_origins="*")

        # 配置路由
        self._setup_routes()

        # 服务器线程
        self.server_thread: Optional[threading.Thread] = None

    def _setup_routes(self):
        """设置路由"""

        @self.app.route('/')
        def index():
            """主页"""
            return render_template('dashboard.html')

        @self.app.route('/api/data')
        def get_data():
            """获取当前数据"""
            return jsonify(self._collect_data())

    def start(self):
        """启动服务器"""
        port = self.config.get('port', 8050)

        self.server_thread = threading.Thread(
            target=lambda: self.socketio.run(
                self.app,
                host='0.0.0.0',
                port=port,
                debug=False,
```

```
                use_reloader=False
            )
        )
        self.server_thread.daemon = True
        self.server_thread.start()

        self.context.logger.info(
            f"可视化服务器已启动: http://localhost:{port}"
        )

    def stop(self):
        """停止服务器"""
        self.context.logger.info("可视化服务器已停止")

    def update_data(self):
        """更新数据并推送到客户端"""
        data = self._collect_data()
        self.socketio.emit('update', data)

    def _collect_data(self) -> Dict[str, Any]:
        """收集当前数据"""
        portfolio = self.context.portfolio
        benchmark_mgr = self.context.benchmark_manager

        # 基本信息
        data = {
            'strategy_name': self.context.strategy_name,
            'mode': self.context.mode,
            'frequency': self.context.frequency,
            'current_dt': self.context.current_dt.isoformat() if self.context.current_dt else None,
            'is_running': self.context.is_running,
        }

        # 账户信息
        data['portfolio'] = {
            'total_value': portfolio.total_value,
            'cash': portfolio.cash,
            'positions_value': sum(
                p.market_value for p in self.context.position_manager.get_all_positions()
            ),
            'returns': portfolio.returns,
        }

        # 基准信息
        if benchmark_mgr:
            data['benchmark'] = {
                'returns': benchmark_mgr.get_current_returns(),
                'value': benchmark_mgr.get_current_value(),
            }

        # 持仓信息
        data['positions'] = [
            {
                'symbol': p.symbol,
                'amount': p.total_amount,
                'avg_cost': p.avg_cost,
                'current_price': p.current_price,
                'market_value': p.market_value,
                'pnl': p.unrealized_pnl,
                'pnl_ratio': p.unrealized_pnl_ratio,
            }
            for p in self.context.position_manager.get_all_positions()
        ]

        # 订单信息（最近10条）
        filled_orders = self.context.order_manager.get_filled_orders()
        data['orders'] = [
            {
                'id': o.id,
                'symbol': o.symbol,
                'side': o.side.value,
                'amount': o.amount,
                'price': o.filled_price,
                'commission': o.commission,
```

```
                'time': o.filled_time.isoformat() if o.filled_time else None,
            }
            for o in filled_orders[-10:]
        ]

        # 历史收益曲线数据
        if hasattr(portfolio, 'history'):
            data['equity_curve'] = portfolio.history

        # 基准收益曲线
        if benchmark_mgr:
            data['benchmark_curve'] = benchmark_mgr.get_benchmark_data()

        return data
```

12.2 前端模板

```
<!-- visualization/templates/dashboard.html -->
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>QTrader - 实时监控</title>
    <script src="https://cdn.socket.io/4.5.4/socket.io.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
    <style>
        * {
            margin: 0;
            padding: 0;
            box-sizing: border-box;
        }

        body {
            font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Arial, sans-serif;
            background-color: #f5f7fa;
            padding: 20px;
        }

        .header {
            background: white;
            padding: 20px 30px;
            border-radius: 8px;
            margin-bottom: 20px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.08);
        }

        .header h1 {
            font-size: 24px;
            color: #333;
            margin-bottom: 8px;
        }

        .header .info {
            color: #666;
            font-size: 14px;
        }

        .header .info span {
            margin-right: 20px;
        }

        .metrics {
            display: grid;
            grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
            gap: 20px;
            margin-bottom: 20px;
        }

        .metric-card {
            background: white;
            padding: 20px;
```

```
            border-radius: 8px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.08);
        }

        .metric-card h3 {
            margin: 0 0 12px 0;
            color: #666;
            font-size: 13px;
            font-weight: 500;
            text-transform: uppercase;
            letter-spacing: 0.5px;
        }

        .metric-card .value {
            font-size: 28px;
            font-weight: 600;
            color: #333;
        }

        .positive { color: #10b981; }
        .negative { color: #ef4444; }
        .neutral { color: #6b7280; }

        .chart-container {
            background: white;
            padding: 20px;
            border-radius: 8px;
            margin-bottom: 20px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.08);
        }

        .chart-container h2 {
            margin: 0 0 20px 0;
            font-size: 18px;
            color: #333;
        }

        .table-container {
            background: white;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.08);
            overflow-x: auto;
            margin-bottom: 20px;
        }

        .table-container h2 {
            margin: 0 0 15px 0;
            font-size: 18px;
            color: #333;
        }

        table {
            width: 100%;
            border-collapse: collapse;
        }

        th, td {
            padding: 12px;
            text-align: left;
            border-bottom: 1px solid #e5e7eb;
        }

        th {
            background-color: #f9fafb;
            font-weight: 600;
            font-size: 13px;
            color: #374151;
            text-transform: uppercase;
            letter-spacing: 0.5px;
        }

        td {
            font-size: 14px;
```

```
            color: #1f2937;
        }

        tr:hover {
            background-color: #f9fafb;
        }

        .status-badge {
            display: inline-block;
            padding: 4px 8px;
            border-radius: 4px;
            font-size: 12px;
            font-weight: 500;
        }

        .status-running {
            background-color: #10b981;
            color: white;
        }

        .status-stopped {
            background-color: #6b7280;
            color: white;
        }

        .empty-state {
            text-align: center;
            padding: 40px;
            color: #9ca3af;
        }
    </style>
</head>
<body>
    <div class="header">
        <h1 id="strategy-name">QTrader</h1>
        <div class="info">
            <span><strong>模式:</strong> <span id="mode"></span></span>
            <span><strong>频率:</strong> <span id="frequency"></span></span>
            <span><strong>当前时间:</strong> <span id="current-time"></span></span>
            <span id="status-badge"></span>
        </div>
    </div>

    <div class="metrics">
        <div class="metric-card">
            <h3>账户总资产</h3>
            <div class="value neutral" id="total-value">¥0</div>
        </div>
        <div class="metric-card">
            <h3>策略收益率</h3>
            <div class="value" id="returns">0%</div>
        </div>
        <div class="metric-card">
            <h3>基准收益率</h3>
            <div class="value" id="benchmark-returns">0%</div>
        </div>
        <div class="metric-card">
            <h3>超额收益</h3>
            <div class="value" id="alpha">0%</div>
        </div>
    </div>

    <div class="chart-container">
        <h2>权益曲线</h2>
        <canvas id="equity-chart"></canvas>
    </div>

    <div class="table-container">
        <h2>当前持仓</h2>
        <table id="positions-table">
            <thead>
                <tr>
                    <th>股票代码</th>
                    <th>持仓数量</th>
```

```
                    <th>平均成本</th>
                    <th>当前价格</th>
                    <th>市值</th>
                    <th>盈亏</th>
                    <th>盈亏率</th>
                </tr>
            </thead>
            <tbody></tbody>
        </table>
        <div id="positions-empty" class="empty-state" style="display:none;">
            暂无持仓
        </div>
    </div>

    <div class="table-container">
        <h2>最近成交</h2>
        <table id="orders-table">
            <thead>
                <tr>
                    <th>时间</th>
                    <th>股票代码</th>
                    <th>方向</th>
                    <th>数量</th>
                    <th>价格</th>
                    <th>手续费</th>
                </tr>
            </thead>
            <tbody></tbody>
        </table>
        <div id="orders-empty" class="empty-state" style="display:none;">
            暂无成交记录
        </div>
    </div>

    <script>
        const socket = io();
        let equityChart = null;

        // 初始化图表
        function initChart() {
            const ctx = document.getElementById('equity-chart').getContext('2d');
            equityChart = new Chart(ctx, {
                type: 'line',
                data: {
                    labels: [],
                    datasets: [
                        {
                            label: '策略收益',
                            data: [],
                            borderColor: '#10b981',
                            backgroundColor: 'rgba(16, 185, 129, 0.1)',
                            tension: 0.4,
                            fill: true
                        },
                        {
                            label: '基准收益',
                            data: [],
                            borderColor: '#3b82f6',
                            backgroundColor: 'rgba(59, 130, 246, 0.1)',
                            tension: 0.4,
                            fill: true
                        }
                    ]
                },
                options: {
                    responsive: true,
                    maintainAspectRatio: false,
                    interaction: {
                        intersect: false,
                        mode: 'index'
                    },
                    plugins: {
                        legend: {
                            position: 'top',
```

```
                    }
                },
                scales: {
                    y: {
                        ticks: {
                            callback: function(value) {
                                return (value * 100).toFixed(2) + '%';
                            }
                        }
                    }
                }
            }
        });
    }

    // 更新页面
    function updatePage(data) {
        // 更新基本信息
        document.getElementById('strategy-name').textContent = data.strategy_name || 'QTrader';
        document.getElementById('mode').textContent = data.mode || '-';
        document.getElementById('frequency').textContent = data.frequency || '-';
        document.getElementById('current-time').textContent = data.current_dt ? new Date(data.current_dt)

        // 更新状态标签
        const statusBadge = document.getElementById('status-badge');
        if (data.is_running) {
            statusBadge.innerHTML = '<span class="status-badge status-running">运行中</span>';
        } else {
            statusBadge.innerHTML = '<span class="status-badge status-stopped">已停止</span>';
        }

        // 更新指标
        if (data.portfolio) {
            document.getElementById('total-value').textContent =
                '¥' + data.portfolio.total_value.toLocaleString('zh-CN', {minimumFractionDigits: 2, maxim

            const returns = data.portfolio.returns * 100;
            const returnsElem = document.getElementById('returns');
            returnsElem.textContent = returns.toFixed(2) + '%';
            returnsElem.className = returns >= 0 ? 'value positive' : 'value negative';
        }

        if (data.benchmark) {
            const benchmarkReturns = data.benchmark.returns * 100;
            const benchmarkElem = document.getElementById('benchmark-returns');
            benchmarkElem.textContent = benchmarkReturns.toFixed(2) + '%';
            benchmarkElem.className = benchmarkReturns >= 0 ? 'value positive' : 'value negative';

            const alpha = (data.portfolio.returns - data.benchmark.returns) * 100;
            const alphaElem = document.getElementById('alpha');
            alphaElem.textContent = alpha.toFixed(2) + '%';
            alphaElem.className = alpha >= 0 ? 'value positive' : 'value negative';
        }

        // 更新图表
        if (data.equity_curve && equityChart) {
            equityChart.data.labels = data.equity_curve.map(d => d.date);
            equityChart.data.datasets[0].data = data.equity_curve.map(d => d.returns);

            if (data.benchmark_curve) {
                equityChart.data.datasets[1].data = data.benchmark_curve.map(d => d.returns);
            }

            equityChart.update('none');
        }

        // 更新持仓表格
        const positionsTable = document.querySelector('#positions-table tbody');
        const positionsEmpty = document.getElementById('positions-empty');

        if (data.positions && data.positions.length > 0) {
            positionsTable.style.display = '';
            positionsEmpty.style.display = 'none';
            positionsTable.innerHTML = '';
```

```
                data.positions.forEach(pos => {
                    const row = positionsTable.insertRow();
                    row.insertCell(0).textContent = pos.symbol;
                    row.insertCell(1).textContent = pos.amount.toLocaleString();
                    row.insertCell(2).textContent = '¥' + pos.avg_cost.toFixed(2);
                    row.insertCell(3).textContent = '¥' + (pos.current_price || 0).toFixed(2);
                    row.insertCell(4).textContent = '¥' + pos.market_value.toLocaleString('zh-CN', {minimumFr

                    const pnlCell = row.insertCell(5);
                    pnlCell.textContent = '¥' + pos.pnl.toLocaleString('zh-CN', {minimumFractionDigits: 2});
                    pnlCell.className = pos.pnl >= 0 ? 'positive' : 'negative';

                    const pnlRatioCell = row.insertCell(6);
                    pnlRatioCell.textContent = (pos.pnl_ratio * 100).toFixed(2) + '%';
                    pnlRatioCell.className = pos.pnl_ratio >= 0 ? 'positive' : 'negative';
                });
            } else {
                positionsTable.style.display = 'none';
                positionsEmpty.style.display = '';
            }

            // 更新订单表格
            const ordersTable = document.querySelector('#orders-table tbody');
            const ordersEmpty = document.getElementById('orders-empty');

            if (data.orders && data.orders.length > 0) {
                ordersTable.style.display = '';
                ordersEmpty.style.display = 'none';
                ordersTable.innerHTML = '';

                data.orders.slice().reverse().forEach(order => {
                    const row = ordersTable.insertRow();
                    row.insertCell(0).textContent = order.time ? new Date(order.time).toLocaleString('zh-CN')
                    row.insertCell(1).textContent = order.symbol;
                    row.insertCell(2).textContent = order.side === 'buy' ? '买入' : '卖出';
                    row.insertCell(3).textContent = order.amount.toLocaleString();
                    row.insertCell(4).textContent = '¥' + (order.price || 0).toFixed(2);
                    row.insertCell(5).textContent = '¥' + (order.commission || 0).toFixed(2);
                });
            } else {
                ordersTable.style.display = 'none';
                ordersEmpty.style.display = '';
            }
        }

        // WebSocket连接
        socket.on('connect', function() {
            console.log('已连接到服务器');

            // 获取初始数据
            fetch('/api/data')
                .then(res => res.json())
                .then(data => updatePage(data))
                .catch(err => console.error('获取数据失败:', err));
        });

        socket.on('update', function(data) {
            updatePage(data);
        });

        socket.on('disconnect', function() {
            console.log('与服务器断开连接');
        });

        // 初始化
        document.addEventListener('DOMContentLoaded', function() {
            initChart();
        });
    </script>
</body>
</html>
```

13. 配置系统详解

13.1 完整配置文件示例

```yaml
# configs/backtest.yaml
# QTrader回测配置文件

# ========== 运行模式 ==========
mode: backtest  # backtest（回测） 或 simulation（模拟盘）

# ========== 策略基本信息 ==========
strategy_name: "MyStrategy"  # 策略名称（可选，默认使用策略类名）

# ========== 运行频率 ==========
frequency: daily  # daily（日频）、minute（分钟频）、tick（Tick频，3秒）

# ========== 回测时间范围 ==========
start_date: "2023-01-01"  # 回测开始日期（YYYY-MM-DD）
end_date: "2023-12-31"    # 回测结束日期（YYYY-MM-DD）

# ========== 初始资金 ==========
initial_cash: 1000000  # 初始资金（元）

# ========== 基准配置 ==========
benchmark:
  symbol: "000300"  # 基准标的代码（'000300'=沪深300）

# ========== 生命周期钩子时间配置 ==========
# handle_bar执行时间（仅日频有效）
# 方式1：单个时间点（默认）
handle_bar_time: "14:55:00"

# 方式2：多个时间点
# handle_bar_times:
#   - "10:00:00"
#   - "14:00:00"
#   - "14:55:00"

broker_settle_time: "15:30:00"   # broker_settle执行时间（默认15:30）

# ========== 撮合配置 ==========
matching:
  # 滑点配置
  slippage:
    type: fixed      # 滑点类型（目前仅支持fixed）
    rate: 0.001      # 固定滑点率（0.1%）

  # 手续费配置
  commission:
    buy_commission: 0.0002    # 买入佣金率（万分之二）
    sell_commission: 0.0002   # 卖出佣金率（万分之二）
    buy_tax: 0.0              # 买入印花税率（A股为0）
    sell_tax: 0.001           # 卖出印花税率（千分之一）
    min_commission: 5.0       # 最低佣金（元）

# ========== 可视化配置 ==========
visualization:
  enable: true          # 是否启用可视化（true/false）
  port: 8050            # Web服务器端口
  update_interval: 1    # 数据更新间隔（秒）

# ========== 日志配置 ==========
logging:
  level: INFO                   # 日志级别（DEBUG/INFO/WARNING/ERROR）
  file: logs/backtest.log       # 日志文件路径
  console_output: true          # 是否输出到控制台

# ========== 状态保存配置 ==========
state:
  auto_save: true       # 是否自动保存状态
  save_dir: .states     # 状态保存目录
```

```yaml
# ========== 报告配置 ==========
report:
  output_dir: reports           # 报告输出目录
  auto_generate: true           # 是否自动生成报告
```

```yaml
# configs/simulation.yaml
# QTrader模拟盘配置文件

# ========== 运行模式 ==========
mode: simulation  # 模拟盘模式

# ========== 运行频率 ==========
frequency: minute  # 模拟盘通常使用minute或tick

# ========== 初始资金 ==========
initial_cash: 100000  # 模拟盘初始资金

# ========== 基准配置 ==========
benchmark:
  symbol: "000300"

# ========== 生命周期钩子时间配置 ==========
handle_bar_time: "14:55:00"      # 仅日频有效
broker_settle_time: "15:30:00"   # broker_settle执行时间

# ========== 撮合配置 ==========
matching:
  slippage:
    type: fixed
    rate: 0.001
  commission:
    buy_commission: 0.0002
    sell_commission: 0.0002
    buy_tax: 0.0
    sell_tax: 0.001
    min_commission: 5.0

# ========== 可视化配置 ==========
visualization:
  enable: true
  port: 8050
  update_interval: 1

# ========== 日志配置 ==========
logging:
  level: INFO
  file: logs/simulation.log
  console_output: true

# ========== 状态保存配置 ==========
state:
  auto_save: true
  save_dir: .states
```

14. 完整使用示例

14.1 简单均线策略示例

```python
# examples/strategies/simple_ma.py

"""
简单双均线策略示例

策略逻辑：
```

```
1. 计算短期和长期均线
2. 金叉买入，死叉卖出
3. 仅持有一只股票

注意：用户需要自行实现历史数据获取
"""

from qtrader import Strategy, Engine
from qtrader.data.examples import StockAPIProvider
from stock_api_sdk import StockAPIClient


class SimpleMAStrategy(Strategy):
    """简单双均线策略"""

    def initialize(self, context):
        """策略初始化"""
        # 设置策略参数
        context.set('ma_short', 5)   # 短期均线周期
        context.set('ma_long', 20)   # 长期均线周期
        context.set('symbol', '000001')  # 交易股票

        # 初始化价格历史（用于计算均线）
        context.set('price_history', [])

        context.logger.info("策略初始化完成")
        context.logger.info(f"交易标的: {context.get('symbol')}")
        context.logger.info(f"短期均线: {context.get('ma_short')}日")
        context.logger.info(f"长期均线: {context.get('ma_long')}日")

    def before_trading(self, context, data):
        """盘前准备"""
        context.logger.info(f"===== {context.current_dt.date()} 盘前准备 =====")

    def handle_bar(self, context, data):
        """盘中策略执行"""
        symbol = context.get('symbol')
        ma_short_period = context.get('ma_short')
        ma_long_period = context.get('ma_long')

        # 获取当前价格（框架提供，用于撮合）
        snapshot = context.data_provider.get_current_snapshot(
            symbol,
            context.current_dt,
            context.frequency
        )

        if snapshot is None:
            context.logger.warning(f"无法获取{symbol}当前价格")
            return

        current_price = snapshot['current_price']

        # 更新价格历史
        price_history = context.get('price_history')
        price_history.append(current_price)

        # 保持最近ma_long个价格
        if len(price_history) > ma_long_period:
            price_history = price_history[-ma_long_period:]
        context.set('price_history', price_history)

        # 如果数据不足，不执行交易
        if len(price_history) < ma_long_period:
            context.logger.debug(f"价格历史数据不足({len(price_history)}/{ma_long_period})")
            return

        # 计算均线
        ma_short = sum(price_history[-ma_short_period:]) / ma_short_period
        ma_long = sum(price_history) / ma_long_period

        context.logger.debug(
            f"当前价格: {current_price:.2f}, "
            f"MA{ma_short_period}: {ma_short:.2f}, "
```

```
                f"MA{ma_long_period}: {ma_long:.2f}"
            )

            # 获取当前持仓
            position = context.position_manager.get_position(symbol)

            # 交易逻辑
            if ma_short > ma_long:
                # 金叉：买入
                if position is None or position.total_amount == 0:
                    # 使用50%资金买入
                    cash = context.portfolio.cash
                    amount = int(cash * 0.5 / current_price / 100) * 100

                    if amount > 0:
                        context.order_manager.submit_market_order(symbol, amount)
                        context.logger.info(
                            f"🔺 金叉买入信号: {symbol} {amount}股 "
                            f"@{current_price:.2f}"
                        )

            elif ma_short < ma_long:
                # 死叉：卖出
                if position and position.total_amount > 0:
                    context.order_manager.submit_market_order(
                        symbol,
                        -position.total_amount
                    )
                    context.logger.info(
                        f"🔻 死叉卖出信号: {symbol} {position.total_amount}股 "
                        f"@{current_price:.2f}"
                    )

        def after_trading(self, context, data):
            """盘后处理"""
            # 打印当日成交情况
            filled_orders = context.order_manager.get_filled_orders_today()
            context.logger.info(f"今日成交订单数: {len(filled_orders)}")

            # 打印账户信息
            portfolio = context.portfolio
            context.logger.info(
                f"账户总资产: ¥{portfolio.total_value:,.2f}, "
                f"收益率: {portfolio.returns:.2%}"
            )

            # 打印基准信息
            benchmark_returns = context.benchmark_manager.get_current_returns()
            context.logger.info(f"基准收益率: {benchmark_returns:.2%}")

        def broker_settle(self, context):
            """日终结算"""
            # 这里可以做一些对账、数据收集等工作
            context.logger.info("日终结算完成")

        def on_end(self, context):
            """策略结束"""
            final_returns = context.portfolio.returns
            benchmark_returns = context.benchmark_manager.get_current_returns()
            alpha = final_returns - benchmark_returns

            context.logger.info("===== 策略运行结束 =====")
            context.logger.info(f"策略最终收益率: {final_returns:.2%}")
            context.logger.info(f"基准最终收益率: {benchmark_returns:.2%}")
            context.logger.info(f"超额收益: {alpha:.2%}")


    if __name__ == '__main__':
        # 创建Stock API客户端
        api_client = StockAPIClient(
            base_url="http://your-api-server/api/v1",
            api_key="your_api_key"
        )
```

```
    # 创建数据提供者
    data_provider = StockAPIProvider(api_client)

    # 创建引擎
    engine = Engine(config_path='configs/backtest.yaml')

    # 运行策略
    engine.run(
        strategy_class=SimpleMAStrategy,
        data_provider=data_provider,
        start_date='2023-01-01',
        end_date='2023-12-31',
        strategy_name='SimpleMA'
    )
```

14.2 多时间点策略示例

```
# examples/strategies/multi_timepoint_strategy.py

"""
多时间点策略示例

策略逻辑：
1. 在不同时间点执行不同的操作
2. 10:00 检查开仓机会
3. 14:00 调整仓位
4. 14:55 平仓检查
"""

from qtrader import Strategy, Engine
from qtrader.data.examples import StockAPIProvider
from stock_api_sdk import StockAPIClient


class MultiTimepointStrategy(Strategy):
    """多时间点策略"""

    def initialize(self, context):
        """策略初始化"""
        context.set('symbol', '000001')
        context.logger.info("多时间点策略初始化完成")

    def handle_bar(self, context, data):
        """盘中策略执行"""
        current_time = context.current_dt.time()
        symbol = context.get('symbol')

        # 获取当前价格
        snapshot = context.data_provider.get_current_snapshot(
            symbol,
            context.current_dt,
            context.frequency
        )

        if snapshot is None:
            return

        current_price = snapshot['current_price']
        position = context.position_manager.get_position(symbol)

        # 根据当前时间执行不同逻辑
        if current_time.hour == 10 and current_time.minute == 0:
            # 10:00 - 检查开仓机会
            context.logger.info("10:00 检查开仓机会")

            if position is None or position.total_amount == 0:
                # 开仓
                cash = context.portfolio.cash
                amount = int(cash * 0.3 / current_price / 100) * 100

                if amount > 0:
```

```
                    context.order_manager.submit_market_order(symbol, amount)
                    context.logger.info(f"开仓: {symbol} {amount}股")

        elif current_time.hour == 14 and current_time.minute == 0:
            # 14:00 - 调整仓位
            context.logger.info("14:00 调整仓位")

            if position and position.total_amount > 0:
                # 加仓
                cash = context.portfolio.cash
                amount = int(cash * 0.2 / current_price / 100) * 100

                if amount > 0:
                    context.order_manager.submit_market_order(symbol, amount)
                    context.logger.info(f"加仓: {symbol} {amount}股")

        elif current_time.hour == 14 and current_time.minute == 55:
            # 14:55 - 平仓检查
            context.logger.info("14:55 平仓检查")

            if position and position.total_amount > 0:
                # 根据盈亏决定是否平仓
                if position.unrealized_pnl_ratio > 0.02:  # 盈利超过2%
                    context.order_manager.submit_market_order(
                        symbol,
                        -position.total_amount
                    )
                    context.logger.info(f"止盈平仓: {symbol}")
                elif position.unrealized_pnl_ratio < -0.01:  # 亏损超过1%
                    context.order_manager.submit_market_order(
                        symbol,
                        -position.total_amount
                    )
                    context.logger.info(f"止损平仓: {symbol}")

    def on_end(self, context):
        """策略结束"""
        final_returns = context.portfolio.returns
        context.logger.info(f"策略最终收益率: {final_returns:.2%}")


if __name__ == '__main__':
    # 注意：需要在配置文件中设置多个时间点
    # handle_bar_times:
    #   - "10:00:00"
    #   - "14:00:00"
    #   - "14:55:00"

    api_client = StockAPIClient(
        base_url="http://your-api-server/api/v1",
        api_key="your_api_key"
    )

    data_provider = StockAPIProvider(api_client)

    engine = Engine(config_path='configs/multi_timepoint.yaml')

    engine.run(
        strategy_class=MultiTimepointStrategy,
        data_provider=data_provider,
        start_date='2023-01-01',
        end_date='2023-12-31',
        strategy_name='MultiTimepoint'
    )
```

15. 实施路线图

15.1 开发阶段划分

阶段一：核心框架（2-3周）

目标：搭建可运行的最小框架

任务清单：

- [x] Context、Engine基础设计
- [x] 数据接口合约定义（AbstractDataProvider，移除get_history_bars）
- [x] 订单、持仓、账户模型（包含即时/历史订单区分）
- [x] 基础撮合引擎（支持即时和历史订单撮合）
- [ ] 单元测试（订单、持仓、账户）
- [ ] 测试日频回测

验收标准：

- 能够运行简单的日频策略
- 订单撮合逻辑正确
- 即时订单和历史订单正确区分

阶段二：多频率与多时间点支持（1-2周）

目标：完善时间调度系统

任务清单：

- [x] Scheduler支持多时间点（日频）
- [x] 分钟频回测
- [x] Tick频回测（带时间校准）
- [ ] 历史订单自动检查机制
- [ ] 订单过期策略
- [ ] 测试各频率下的订单撮合

验收标准：

- 日频支持多时间点配置
- 分钟频和Tick频正常运行
- 历史订单在每次handle_bar时正确检查
- 日频未成交订单正确过期

阶段三：数据提供者示例（1周）

目标：提供完整的数据提供者实现示例

任务清单：

- [x] 基于Stock API的实现
- [x] 本地缓存优化
- [ ] 基于本地CSV的实现
- [ ] 使用文档
- [ ] 数据提供者开发指南

验收标准：

- 用户可以参考示例快速实现自己的数据提供者
- 缓存机制有效减少API调用

- 文档清晰完整

阶段四：基准与分析（1周）

目标：完善基准跟踪和绩效分析

任务清单：

- [x] BenchmarkManager
- [ ] PerformanceAnalyzer（夏普比率、最大回撤等）
- [ ] RiskAnalyzer（波动率、贝塔等）
- [ ] ReportGenerator（生成HTML报告）

验收标准：

- 基准自动跟踪
- 绩效指标计算准确
- 报告美观实用

阶段五：可视化系统（1-2周）

目标：实现Web实时可视化

任务清单：

- [x] VisualizationServer设计
- [x] 前端模板
- [ ] WebSocket推送实现
- [ ] 多图表支持（K线图、持仓分布等）
- [ ] 测试回测与模拟盘

验收标准：

- Web界面实时更新
- 图表美观流畅
- 回测和模拟盘都能正常显示

阶段六：状态管理与模拟盘（1周）

目标：完善状态持久化和模拟盘功能

任务清单：

- [x] StateSerializer
- [ ] 自动保存点
- [ ] 恢复测试
- [ ] 模拟盘时间推进
- [ ] broker_settle功能测试

验收标准：

- 状态可以正确保存和恢复
- 模拟盘实时运行稳定
- broker_settle可用于对账

阶段七：文档与示例（1周）

目标：完善文档和示例

任务清单：

- [ ] 用户指南
- [ ] API文档
- [ ] 策略示例（至少5个）
- [ ] 数据提供者开发指南
- [ ] FAQ

验收标准：

- 新用户可以通过文档快速上手
- 所有API都有清晰的说明
- 示例代码可以直接运行

阶段八：测试与优化（1-2周）

目标：全面测试和性能优化

任务清单：

- [ ] 单元测试覆盖率 > 80%
- [ ] 集成测试
- [ ] 端到端测试
- [ ] 性能测试（百万级数据）
- [ ] 内存优化
- [ ] Bug修复

验收标准：

- 所有测试通过
- 回测速度：日频 > 1000天/秒，分钟频 > 100天/秒
- 无内存泄漏

15.2 开发优先级

P0（必须）：

- 核心撮合引擎（即时/历史订单区分）
- 基础时间调度
- 订单、持仓、账户管理
- 数据接口合约
- 日频回测

P1（重要）：

- 多时间点支持
- 分钟频/Tick频
- 基准管理
- 基本可视化
- 数据提供者示例

P2（建议）：

- 高级可视化
- 完整绩效分析
- 状态持久化
- broker_settle功能

P3（可选）：

- 复杂订单类型
- 更多示例策略
- 性能优化工具

15.3 版本规划

v1.0.0（MVP）（预计3-4周）：

- 核心回测功能
- 日频单时间点
- 基本撮合引擎
- 简单示例

v1.1.0（预计+2周）：

- 多时间点支持
- 分钟频/Tick频
- 历史订单智能撮合
- 数据提供者示例

v1.2.0（预计+2周）：

- 基准管理
- 基本可视化
- 绩效分析

v2.0.0（预计+3周）：

- 完整可视化
- 模拟盘支持
- broker_settle功能
- 状态持久化

v2.1.0及以后：

- 性能优化
- 更多示例
- 高级功能

附录：关键设计决策总结

1. 数据接口简化

决策：移除 `get_history_bars` 接口

理由：

- 框架只关心交易，不关心因子计算
- 历史数据获取是策略的职责，不是框架的职责
- 简化框架与数据层的耦合

影响：

- 用户需要自行实现历史数据获取逻辑
- 框架更加轻量级
- 数据提供者实现更简单

2. 订单区分即时和历史

决策：订单分为即时订单和历史订单，撮合逻辑不同

理由：

- 保证回测的因果一致性
- 即时订单可以使用当前时刻的市场状态
- 历史订单不能使用未来的市场状态

影响：

- 撮合引擎复杂度增加
- 回测结果更真实
- 避免未来函数

3. 支持日频多时间点

决策：日频策略支持配置多个handle_bar执行时间点

理由：

- 满足复杂策略需求（开仓、调仓、平仓在不同时间）
- 每个时间点都会检查历史订单
- 增加策略灵活性

影响：

- 调度器逻辑更复杂
- 配置更灵活
- 用户可以在一天内多次调整仓位

4. broker_settle暴露接口

决策：在broker_settle中允许用户修正持仓和账户

理由：

- 模拟盘需要与实盘对账
- 用户可能需要根据实盘调整模拟盘状态
- 提供数据准备的时间窗口

影响：

- 用户有更大的控制权
- 可以实现模拟盘与实盘的同步
- 需要用户谨慎使用，避免破坏数据一致性

文档版本: 5.0.0（完整可交付版本）
最后更新: 2025年10月
状态: 完整设计，可开始实施