A股量化交易框架设计文档 (QTrader Framework)

简称: QTrader (Quantitative Trading Framework for A-shares)

目录

---

1. 系统概述

1.1 设计目标

构建一个**轻量级、高度解耦、专注于交易逻辑**的A股量化交易框架，具备以下特点：

- **数据完全解耦**：框架不自带数据系统，用户通过工具层自定义数据获取方式
- **回测/模拟统一**：同一策略代码可无缝切换回测与模拟盘模式
- **事件驱动架构**：清晰的生命周期钩子（初始化、盘前、盘中、盘后）
- **精准撮合机制**：基于真实市场规则，支持限价单和市价单
- **实时可视化**：Web界面实时展示回测/模拟盘运行状态
- **基准对比**：支持任意标的作为基准，自动对比收益
- **多频率支持**：日频、分钟频、Tick频（3秒）
- **状态可持久化**：支持暂停/恢复

1.2 核心差异化特性

特性	QTrader	Zipline	Backtrader	QFF
数据获取	用户自定义工具层	预下载Bundle	需要Feeds	MongoDB预存
数据依赖	仅交易必需数据	全量历史数据	全量历史数据	全量历史数据
撮合机制	ask1/bid1真实撮合	简化撮合	Bar撮合	分钟/日线撮合
实时可视化	内置Web界面	无	无	无
框架职责	纯交易逻辑	数据+交易	数据+交易	数据+交易
基准管理	自动跟踪任意标的	固定基准	手动实现	固定基准

1.3 框架职责边界

框架负责：

- 时间推进与事件调度
- 订单管理与撮合成交
- 账户与持仓管理
- 基准跟踪与对比
- 绩效统计与风险分析
- 实时可视化展示

框架不负责（用户实现）：

- 数据获取与存储
- 因子计算
- 选股逻辑
- 实盘下单（仅模拟）

---

2. 核心设计理念

2.1 架构原则

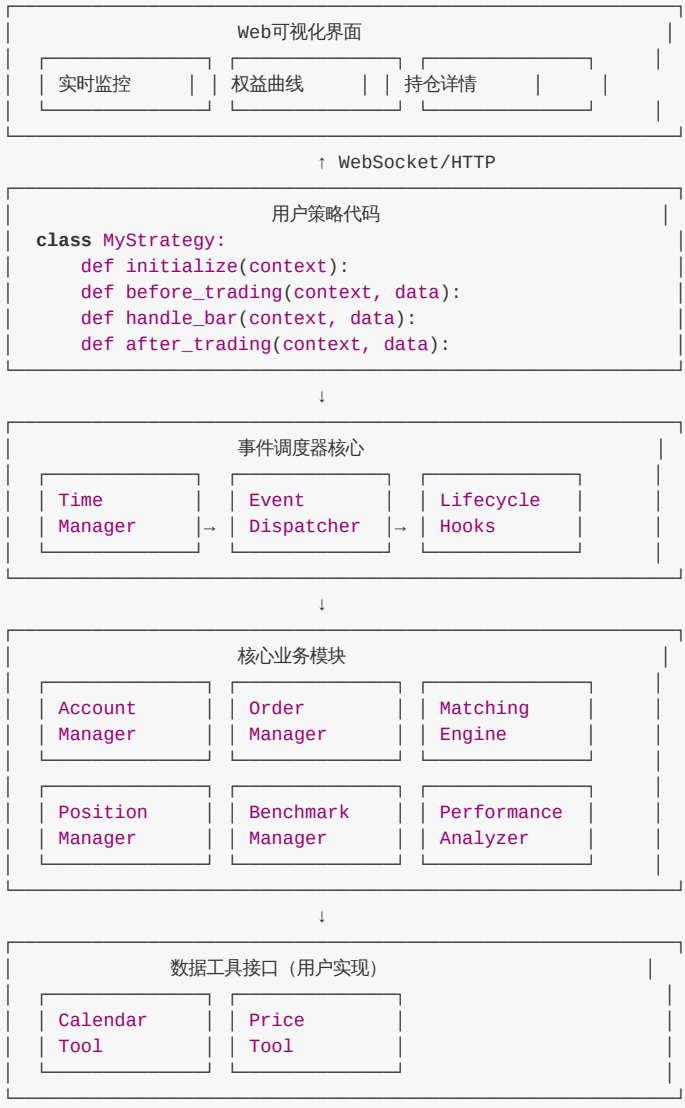
1. **框架与数据解耦**
    - 框架只定义数据接口规范
    - 用户通过工具层实现具体数据获取
    - 框架仅关心：交易日历、成交价格数据、基准数据
2. **事件驱动**
    - 核心是事件调度器
    - 策略通过回调函数响应事件
3. **职责单一**
    - 框架专注交易引擎
    - 其他功能（数据、因子）由用户或第三方库提供
4. **实时可见**
    - Web界面实时展示运行状态
    - 支持回测与模拟盘

2.2 分层架构

```
│  - 可视化服务 (Visualization Server)      │
└──────────────────────────────────────────┘
                    ↓
┌──────────────────────────────────────────┐
│    数据工具层 (Data Tools - 用户实现)      │
│  - 交易日历工具                           │
│  - 价格数据工具                           │
└──────────────────────────────────────────┘
```

3. 系统架构设计

3.1 整体架构图



3.2 目录结构

```
qtrader/
├── core/                      # 核心框架
│   ├── __init__.py
│   ├── context.py             # 全局上下文
│   ├── engine.py              # 主引擎
```

```
│   ├── scheduler.py          # 事件调度器
│   ├── time_manager.py       # 时间管理器
│   └── config.py             # 配置管理
│
├── trading/                   # 交易核心
│   ├── __init__.py
│   ├── account.py            # 账户管理
│   ├── 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
│   └── benchmark_tracker.py  # 基准追踪器
│
├── data_tools/                # 数据工具接口
│   ├── __init__.py
│   ├── base.py               # 基类定义
│   ├── calendar.py           # 交易日历接口
│   ├── price.py              # 价格数据接口
│   └── examples/             # 示例实现
│       ├── api_calendar.py
│       ├── api_price.py
│       └── local_price.py
│
├── strategy/                  # 策略相关
│   ├── __init__.py
│   ├── base.py               # 策略基类
│   ├── context.py            # 策略上下文
│   └── api.py                # 策略API
│
├── 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/
│       ├── js/
│       └── charts.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
│   ├── default.yaml
│   ├── backtest.yaml
│   └── simulation.yaml
│
├── examples/                  # 示例
│   ├── strategies/           # 策略示例
│   │   ├── simple_ma.py
│   │   └── mean_reversion.py
│   └── data_tools/           # 数据工具示例
│       ├── stock_api_impl.py
│       └── local_csv_impl.py
│
├── tests/                     # 测试
```

```
│   ├── unit/
│   ├── integration/
│   └── e2e/
│
├── setup.py
├── requirements.txt
└── README.md
```

4. 核心模块详细设计

4.1 Context（全局上下文）

```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"""

    # 基础信息
    mode: str = 'backtest'  # 'backtest' 或 'simulation'
    strategy_name: str = ''
    start_date: str = ''
    end_date: str = ''
    current_dt: Optional[datetime] = None

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

    # 账户相关
    portfolio: Optional['Portfolio'] = None

    # 订单与持仓
    order_manager: Optional['OrderManager'] = None
    position_manager: Optional['PositionManager'] = None

    # 基准管理
    benchmark_manager: Optional['BenchmarkManager'] = None

    # 配置
    config: Dict[str, Any] = field(default_factory=dict)

    # 用户自定义数据存储
    user_data: Dict[str, Any] = field(default_factory=dict)

    # 运行状态
    is_running: bool = False
    is_paused: bool = False

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

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

    # 数据工具（用户注入）
    calendar_tool: Optional['CalendarTool'] = None
    price_tool: Optional['PriceTool'] = 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"""
        return self.user_data.get(key, default)

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

    def register_data_tools(
        self,
        calendar_tool,
        price_tool
    ):
        """注册数据工具（用户调用）"""
        self.calendar_tool = calendar_tool
        self.price_tool = price_tool
```

4.2 Engine（主引擎）

```
# core/engine.py

from typing import Type, Optional
import yaml
from datetime import datetime
from .context import Context
from .scheduler import Scheduler
from .time_manager import TimeManager
from ..trading.account import Portfolio
from ..trading.order_manager import OrderManager
from ..trading.position_manager import PositionManager
from ..trading.matching_engine import MatchingEngine
from ..benchmark.benchmark_manager import BenchmarkManager
from ..visualization.server import VisualizationServer
from ..strategy.base import Strategy
from ..utils.logger import Logger

class Engine:
    """框架主引擎"""

    def __init__(self, config_path: Optional[str] = None):
        # 加载配置
        self.config = self._load_config(config_path)

        # 初始化日志
        self.logger = Logger(self.config.get('logging', {}))

        # 初始化上下文
        self.context = Context(
            mode=self.config['mode'],
            frequency=self.config['frequency'],
            config=self.config,
            logger=self.logger
        )

        # 初始化组件
        self._init_components()

    def _load_config(self, config_path: Optional[str]) -> dict:
        """加载配置文件"""
        if config_path is None:
            config_path = 'configs/default.yaml'

        with open(config_path, 'r', encoding='utf-8') as f:
            config = yaml.safe_load(f)

        return config

    def _init_components(self):
        """初始化所有核心组件"""
```

```
        # 时间管理器
        self.time_manager = TimeManager(self.context)

        # 账户
        self.context.portfolio = Portfolio(
            initial_cash=self.config['initial_cash']
        )

        # 订单与持仓
        self.context.order_manager = OrderManager(self.context)
        self.context.position_manager = PositionManager(self.context)

        # 基准管理器
        self.context.benchmark_manager = BenchmarkManager(
            self.context,
            self.config.get('benchmark', {})
        )

        # 撮合引擎
        self.matching_engine = MatchingEngine(
            self.context,
            self.config['matching']
        )

        # 调度器
        self.scheduler = Scheduler(
            self.context,
            self.time_manager,
            self.matching_engine
        )

        # 可视化服务器（可选）
        if self.config.get('visualization', {}).get('enable', False):
            self.context.visualization_server = VisualizationServer(
                self.context,
                self.config['visualization']
            )

    def run(
        self,
        strategy_class: Type[Strategy],
        calendar_tool,
        price_tool,
        **kwargs
    ):
        """
        运行策略

        Args:
            strategy_class: 策略类
            calendar_tool: 交易日历工具实例
            price_tool: 价格数据工具实例
            **kwargs: 其他参数
        """
        # 注册数据工具
        self.context.register_data_tools(calendar_tool, price_tool)

        # 验证数据工具
        self._validate_data_tools()

        # 实例化策略
        strategy = strategy_class()

        # 注册生命周期钩子
        self.scheduler.register_strategy(strategy)

        # 设置时间范围
        self.context.start_date = kwargs.get('start_date', self.config.get('start_date'))
        self.context.end_date = kwargs.get('end_date', self.config.get('end_date'))
        self.context.strategy_name = kwargs.get('strategy_name', strategy.__class__.__name__)

        # 初始化基准
        benchmark_symbol = self.config.get('benchmark', {}).get('symbol', '000300')
        self.context.benchmark_manager.initialize(benchmark_symbol)
```

```
        # 启动可视化服务器
        if self.context.visualization_server:
            self.context.visualization_server.start()

        # 启动调度器
        self.context.is_running = True
        self.logger.info(f"开始运行策略: {self.context.strategy_name}")
        self.logger.info(f"模式: {self.context.mode}, 频率: {self.context.frequency}")
        self.logger.info(f"基准: {benchmark_symbol}")

        try:
            self.scheduler.run()
        except KeyboardInterrupt:
            self.logger.warning("用户中断运行")
            self.pause()
        except Exception as e:
            self.logger.error(f"运行出错: {e}", exc_info=True)
            raise
        finally:
            self.context.is_running = False

            # 停止可视化服务器
            if self.context.visualization_server:
                self.context.visualization_server.stop()

    def _validate_data_tools(self):
        """验证数据工具是否已注册"""
        if self.context.calendar_tool is None:
            raise RuntimeError("未注册交易日历工具")

        if self.context.price_tool is None:
            raise RuntimeError("未注册价格数据工具")

    def pause(self):
        """暂停运行"""
        self.context.is_paused = True
        self.logger.info("暂停运行，正在保存状态...")
        self._save_state()

    def resume(self, state_path: str):
        """从保存点恢复"""
        self.logger.info(f"从 {state_path} 恢复运行")
        self._load_state(state_path)
        self.context.is_paused = False
        self.scheduler.run()

    def _save_state(self):
        """保存当前状态"""
        from ..utils.serializer import StateSerializer
        serializer = StateSerializer(self.context)
        serializer.save()

    def _load_state(self, path: str):
        """加载状态"""
        from ..utils.serializer import StateSerializer
        serializer = StateSerializer(self.context)
        serializer.load(path)
```

4.3 Scheduler（事件调度器）

关键优化：处理Tick频率超时问题

```
# core/scheduler.py

from datetime import datetime, time, timedelta
from typing import Optional
import time as time_module
from .context import Context
```

```python
from .time_manager import TimeManager
from ..trading.matching_engine import MatchingEngine
from ..strategy.base import Strategy

class Scheduler:
    """事件调度器"""

    # 定义交易时段
    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
    ):
        self.context = context
        self.time_manager = time_manager
        self.matching_engine = matching_engine
        self.strategy: Optional[Strategy] = None

        # 根据频率构建时间点列表
        if context.frequency == 'minute':
            self.time_points = self._build_minute_schedule()
        elif context.frequency == 'tick':
            self.time_points = self._build_tick_schedule()
        else:
            self.time_points = []

        # Tick频率的上次执行时间（用于超时处理）
        self.last_tick_time: Optional[datetime] = None

    def register_strategy(self, strategy: Strategy):
        """注册策略"""
        self.strategy = strategy

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

    def _run_backtest(self):
        """回测模式运行"""
        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._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._call_on_end()

    def _run_simulation(self):
        """模拟盘模式运行"""
        # 调用初始化
        self._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._call_before_trading(now)

            elif self._should_run_handle_bar(now):
                # 执行handle_bar
                self._call_handle_bar(now)
                self.matching_engine.match_orders(now)

                # 更新可视化
                self._update_visualization()

            elif self._should_run_after_trading(now):
                self._call_after_trading(now)
                self.matching_engine.settle()
                self.context.logger.info("当日结算完成")

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

                # 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）"""
        dt = datetime.strptime(f"{date} 09:30:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt

        # 盘前
        self._call_before_trading(dt)

        # 盘中
        self._call_handle_bar(dt)

        # 撮合
        self.matching_engine.match_orders(dt)

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

        # 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）"""
```

```
        dt_before = datetime.strptime(f"{date} 09:25:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt_before
        self._call_before_trading(dt_before)

        # 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._call_handle_bar(dt)
            self.matching_engine.match_orders(dt)

            if self.context.is_paused:
                break

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

        # 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）"""
        dt_before = datetime.strptime(f"{date} 09:25:00", "%Y-%m-%d %H:%M:%S")
        self.context.current_dt = dt_before
        self._call_before_trading(dt_before)

        # 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._call_handle_bar(dt)
            self.matching_engine.match_orders(dt)

            if self.context.is_paused:
                break

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

        # 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 _call_initialize(self):
        """调用初始化钩子"""
        if hasattr(self.strategy, 'initialize'):
            self.context.logger.info("调用策略初始化")
            self.strategy.initialize(self.context)

    def _call_before_trading(self, dt: datetime):
        """调用盘前钩子"""
        if hasattr(self.strategy, 'before_trading'):
            data = self._get_data_proxy(dt)
            self.strategy.before_trading(self.context, data)

    def _call_handle_bar(self, dt: datetime):
        """调用盘中钩子"""
        if hasattr(self.strategy, 'handle_bar'):
            data = self._get_data_proxy(dt)
            self.strategy.handle_bar(self.context, data)

    def _call_after_trading(self, dt: datetime):
        """调用盘后钩子"""
        if hasattr(self.strategy, 'after_trading'):
            data = self._get_data_proxy(dt)
            self.strategy.after_trading(self.context, data)

    def _call_on_end(self):
        """调用结束钩子"""
        if hasattr(self.strategy, 'on_end'):
            self.context.logger.info("调用策略结束钩子")
            self.strategy.on_end(self.context)

    def _get_data_proxy(self, dt: datetime):
        """获取数据代理对象"""
        from ..strategy.data_proxy import DataProxy
        return DataProxy(self.context, dt)

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

    def _should_run_handle_bar(self, now: datetime) -> bool:
        """判断是否应该执行盘中"""
        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：在开盘时执行
        if self.context.frequency == 'daily':
```

```
            return current_time >= self.MORNING_START and current_time < time(9, 31)

        # 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（处理超时）
        elif self.context.frequency == 'tick':
            if self.last_tick_time is None:
                self.last_tick_time = now
                return True

            # 如果距离上次执行超过3秒，立即执行
            elapsed = (now - self.last_tick_time).total_seconds()
            if elapsed >= 3:
                self.last_tick_time = now
                return True

            # 否则等待到3秒整数倍
            return now.second % 3 == 0

        return False

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

    def _sleep_until_next_event(self):
        """休眠到下一个事件点"""
        if self.context.frequency == 'tick':
            # Tick频：休眠1秒后检查（避免错过执行时机）
            time_module.sleep(1)
        elif self.context.frequency == 'minute':
            # 分钟频：休眠到下一分钟
            now = datetime.now()
            time_module.sleep(max(60 - now.second, 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()
```

5. 数据工具层设计

5.1 交易日历工具接口

```
# data_tools/calendar.py

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

class CalendarTool(ABC):
    """交易日历工具基类"""

    @abstractmethod
    def is_trading_day(self, date: str) -> bool:
        """判断是否为交易日"""
        pass

    @abstractmethod
    def get_trading_days(self, start: str, end: str) -> List[str]:
        """获取时间范围内的所有交易日"""
        pass
```

```python
    @abstractmethod
    def get_previous_trading_day(self, date: str, n: int = 1) -> Optional[str]:
        """获取前N个交易日"""
        pass

    @abstractmethod
    def get_next_trading_day(self, date: str, n: int = 1) -> Optional[str]:
        """获取后N个交易日"""
        pass
```

5.2 价格数据工具接口

关键改进：一次性返回所有必需数据

```python
# data_tools/price.py

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

class PriceTool(ABC):
    """价格数据工具基类"""

    @abstractmethod
    def get_price_data(
        self,
        symbol: str,
        dt: datetime,
        frequency: str
    ) -> Optional[Dict]:
        """
        获取指定时刻的完整价格数据（一次性返回所有必需数据）

        Args:
            symbol: 股票代码（不带后缀）
            dt: 时间点
            frequency: 频率 'daily', 'minute', 'tick'

        Returns:
            价格数据字典，格式:
            {
                'current_price': float,  # 当前价格（必需）
                'ask1': float,           # 卖一价（可选）
                'bid1': float,           # 买一价（可选）
                'high_limit': float,     # 涨停价（必需）
                'low_limit': float,      # 跌停价（必需）
                'is_suspended': bool,    # 是否停牌（必需）
            }

            如果数据不存在，返回None
        """
        pass

    @abstractmethod
    def get_close_price(
        self,
        symbol: str,
        date: str
    ) -> Optional[float]:
        """
        获取收盘价（用于基准和结算）

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

        Returns:
            收盘价，如果不存在返回None
```

```
        """
        pass
```

5.3 示例实现：基于Stock API

```python
# data_tools/examples/api_price.py

from typing import Dict, Optional
from datetime import datetime
from ..price import PriceTool

class APIPriceTool(PriceTool):
    """基于Stock API的价格数据工具实现"""

    def __init__(self, api_client, enable_cache: bool = True):
        self.api_client = api_client
        self.enable_cache = enable_cache
        self._cache = {} if enable_cache else None

    def get_price_data(
        self,
        symbol: str,
        dt: datetime,
        frequency: str
    ) -> Optional[Dict]:
        """获取完整价格数据（一次性获取所有必需数据）"""
        # 检查缓存
        if self.enable_cache:
            cache_key = f"{symbol}_{dt.isoformat()}_{frequency}"
            if cache_key in self._cache:
                return self._cache[cache_key]

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

            # 1. 获取价格和五档数据
            if frequency == 'tick':
                tick_data = self._get_tick_data(symbol, dt)
                if tick_data:
                    result['current_price'] = tick_data.get('current_price')
                    result['ask1'] = tick_data.get('ask1_price')
                    result['bid1'] = tick_data.get('bid1_price')
            else:
                kline_data = self._get_kline_data(symbol, dt, frequency)
                if kline_data:
                    result['current_price'] = kline_data.get('current_price')
                    result['ask1'] = None
                    result['bid1'] = None

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

            # 2. 获取涨跌停价格
            limit_prices = self.api_client.market.get_limit_price(
                code=symbol,
                date=date_str
            )
            if limit_prices and symbol in limit_prices:
                limit_data = limit_prices[symbol]
                result['high_limit'] = limit_data.get('high_limit')
                result['low_limit'] = limit_data.get('low_limit')
            else:
                # 无涨跌停数据，设为None
                result['high_limit'] = None
                result['low_limit'] = None

            # 3. 判断停牌
            suspend_result = self.api_client.stock.is_suspended(
```

```
                code=symbol,
                date=date_str
            )
            result['is_suspended'] = suspend_result[0].get('is_suspended', False) if suspend_result else Fals

            # 写入缓存
            if self.enable_cache:
                self._cache[cache_key] = result

            return result

        except Exception as e:
            print(f"获取价格数据失败 {symbol} @ {dt}: {e}")
            return None

    def get_close_price(self, symbol: str, date: str) -> Optional[float]:
        """获取收盘价"""
        try:
            # 使用历史K线接口获取收盘价
            kline_data = self.api_client.market.get_kline(
                code=symbol,
                period='1d',
                end_time=date,
                count=1
            )

            if kline_data and symbol in kline_data and len(kline_data[symbol]) > 0:
                return kline_data[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]:
        """从Tick API获取数据"""
        date_str = dt.strftime('%Y-%m-%d %H:%M:%S')
        tick_data = self.api_client.market.get_tick(code=symbol, date=date_str)

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

        return None

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

        kline_data = self.api_client.market.get_single_vkline(
            code=symbol,
            period=period,
            date=date_str
        )

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

        return None
```

6. 交易撮合机制

6.1 撮合引擎设计

关键修正：

1. **限价单撮合**：以ask1/bid1成交，而非限价

2. **涨跌停判断**：当前价格等于涨跌停价时拒绝

```python
# trading/matching_engine.py

from datetime import datetime
from typing import Optional, Dict
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"""

    def __init__(self, context: Context, config: dict):
        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"""
        order_manager = self.context.order_manager
        open_orders = order_manager.get_open_orders()

        for order in open_orders:
            self._try_match_order(order, dt)

    def _try_match_order(self, order: Order, dt: datetime):
        """尝试撮合单个订单"""
        # 获取完整价格数据（一次性获取）
        price_data = self.context.price_tool.get_price_data(
            order.symbol,
            dt,
            self.context.frequency
        )

        if price_data is None:
            self.context.logger.warning(f"无法获取 {order.symbol} 价格数据，跳过撮合")
            return

        # 检查停牌
        if price_data.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

        # 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, price_data):
            order.reject("触及涨跌停")
            self.context.logger.info(f"订单 {order.id} 拒绝：{order.symbol} 触及涨跌停")
            return

        # 确定撮合价格
        match_price = self._determine_match_price(order, price_data)
        if match_price is None:
            # 限价单无法成交
            return

        # 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

        # 再次检查涨跌停（加滑点后）
        if not self._check_limit_price_range(final_price, price_data, order.side):
            order.reject("加滑点后触及涨跌停")
```

```
            self.context.logger.info(f"订单 {order.id} 拒绝：加滑点后触及涨跌停")
            return

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

        # 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

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

    def _determine_match_price(
        self,
        order: Order,
        price_data: Dict
    ) -> Optional[float]:
        """
        确定撮合价格

        关键逻辑：
        - 市价单：直接使用ask1/bid1（或current_price）
        - 限价单：检查能否成交，成交价为ask1/bid1（或current_price），而非限价
        """
        current_price = price_data.get('current_price')
        ask1 = price_data.get('ask1')
        bid1 = price_data.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:
                    return None  # 无法成交
            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:
                    return None  # 无法成交

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

        关键逻辑：
        - 当前价格 == 涨停价，拒绝买入
        - 当前价格 == 跌停价，拒绝卖出
        """
        current_price = price_data.get('current_price')
        high_limit = price_data.get('high_limit')
        low_limit = price_data.get('low_limit')

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

```
        # 买入：当前价格等于涨停价，拒绝
        if order.side == OrderSide.BUY:
            if abs(current_price - high_limit) < 0.01:  # 浮点数比较
                return False
        # 卖出：当前价格等于跌停价，拒绝
        else:
            if abs(current_price - low_limit) < 0.01:
                return False

        return True

    def _check_limit_price_range(
        self,
        price: float,
        price_data: Dict,
        side: OrderSide
    ) -> bool:
        """检查价格是否超出涨跌停范围"""
        high_limit = price_data.get('high_limit')
        low_limit = price_data.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"""
        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):
                return position.available_amount >= order.amount
            else:
                return position.total_amount >= order.amount

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

        # 更新账户
        portfolio = self.context.portfolio
        position_manager = self.context.position_manager
```

```
        if order.side == OrderSide.BUY:
            # 买入
            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}"
            )
        else:
            # 卖出
            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}"
            )

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

    def settle(self):
        """每日收盘结算"""
        self.context.logger.info("开始每日结算")

        # 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._get_close_price(position.symbol)
            if close_price:
                position.update_price(close_price)

        # 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

        # 记录账户历史
        portfolio.record_history()

        # 更新持仓的可用数量（T+1处理）
        for position in position_manager.get_all_positions():
            if self._is_t_plus_1(position.symbol):
                position.available_amount = position.total_amount

        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"""
        # ETF代码通常5开头为T+0，其他为T+1
        return not symbol.startswith('5')

    def _get_close_price(self, symbol: str) -> Optional[float]:
        """获取当日收盘价"""
```

```
        return self.context.price_tool.get_close_price(
            symbol,
            self.context.current_dt.strftime('%Y-%m-%d')
        )
```

7. 时间与调度系统

7.1 TimeManager

```python
# core/time_manager.py

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

class TimeManager:
    """时间管理器"""

    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):
        self.context = context

    def is_trading_day(self, dt: datetime) -> bool:
        """判断是否为交易日"""
        if self.context.calendar_tool is None:
            raise RuntimeError("未注册交易日历工具")

        date_str = dt.strftime('%Y-%m-%d')
        return self.context.calendar_tool.is_trading_day(date_str)

    def is_trading_time(self, dt: datetime) -> bool:
        """判断是否在交易时段"""
        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"""
        if self.context.calendar_tool is None:
            raise RuntimeError("未注册交易日历工具")

        return self.context.calendar_tool.get_trading_days(start, end)
```

8. 基准管理系统

8.1 BenchmarkManager

```python
# benchmark/benchmark_manager.py

from typing import Optional, Dict, List
from datetime import datetime
```

```python
from ..core.context import Context

class BenchmarkManager:
    """
    基准管理器

    负责：
    1. 跟踪基准标的的每日收盘价
    2. 计算基准收益率
    3. 提供基准对比数据
    """

    def __init__(self, context: Context, config: Dict):
        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 = self.context.start_date
        self.initial_value = self.context.price_tool.get_close_price(
            symbol,
            start_date
        )

        if self.initial_value is None:
            self.context.logger.warning(f"无法获取基准 {symbol} 的初始价格")
            self.initial_value = 1000.0  # 默认值

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

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

        获取当日收盘价，计算收益率
        """
        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.price_tool.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
```

```
        # 记录历史
        self.benchmark_history.append({
            'date': current_date,
            'close_price': close_price,
            'returns': returns,
            'value': self.initial_value * (1 + returns)
        })

        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
```

9. 状态管理与持久化

```
# 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"""

    def __init__(self, context: Context, save_dir: str = '.states'):
        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"""
        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"""
        with open(file_path, 'rb') as f:
            state = pickle.load(f)

        # 恢复状态
        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']

        # 恢复账户
        self.context.portfolio = state['portfolio']

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

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

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

        # 恢复用户数据
        self.context.user_data = state['user_data']

        self.context.logger.info(f"状态已从 {file_path} 加载")
```

10. 回测与模拟盘模式

10.1 配置示例

```
# configs/backtest.yaml
mode: backtest
frequency: daily  # daily | minute | tick
start_date: "2023-01-01"
end_date: "2023-12-31"
initial_cash: 1000000

# 基准配置
benchmark:
  symbol: "000300"  # 沪深300

# 撮合配置
matching:
  slippage:
```

```
    type: fixed
    rate: 0.001
  commission:
    buy_commission: 0.0002
    sell_commission: 0.0002
    sell_tax: 0.001
    min_commission: 5.0

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

# 日志配置
logging:
  level: INFO
  file: logs/backtest.log
```

```
# configs/simulation.yaml
mode: simulation
frequency: minute
initial_cash: 1000000

benchmark:
  symbol: "000300"

matching:
  slippage:
    type: fixed
    rate: 0.001
  commission:
    buy_commission: 0.0002
    sell_commission: 0.0002
    sell_tax: 0.001
    min_commission: 5.0

visualization:
  enable: true
  port: 8050
  update_interval: 1

logging:
  level: INFO
  file: logs/simulation.log
```

11. 实时可视化系统

11.1 设计理念

基于Web的实时可视化界面，参考template.html的设计，提供：

1. **实时监控面板**：
   - 策略运行状态
   - 当前时间与进度
   - 账户总资产
   - 持仓概览
2. **权益曲线图**：
   - 策略收益曲线

- 基准收益曲线
- 实时更新

3. **详细数据表格**：

- 持仓详情
- 订单记录
- 账户历史

11.2 VisualizationServer

```python
# visualization/server.py

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

class VisualizationServer:
    """可视化服务器"""

    def __init__(self, context: Context, config: Dict):
        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"""
        # Flask没有优雅停止方法，这里只是标记
        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.total_value - portfolio.initial_cash) / portfolio.initial_cash,
        }

        # 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,
                'price': p.current_price,
                'value': p.market_value,
                'pnl': p.unrealized_pnl,
            }
            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,
                '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'] = [
                {
                    'date': h['date'],
                    'value': h['total_value'],
```

```
                    'returns': (h['total_value'] - portfolio.initial_cash) / portfolio.initial_cash,
                }
                for h in portfolio.history
            ]

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

        return data
```

11.3 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>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 20px;
            background-color: #f5f5f5;
        }
        .header {
            background: white;
            padding: 20px;
            border-radius: 8px;
            margin-bottom: 20px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.1);
        }
        .metrics {
            display: grid;
            grid-template-columns: repeat(4, 1fr);
            gap: 20px;
            margin-bottom: 20px;
        }
        .metric-card {
            background: white;
            padding: 20px;
            border-radius: 8px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.1);
        }
        .metric-card h3 {
            margin: 0 0 10px 0;
            color: #666;
            font-size: 14px;
        }
        .metric-card .value {
            font-size: 24px;
            font-weight: bold;
            color: #333;
        }
        .positive { color: #4CAF50; }
        .negative { color: #f44336; }

        .chart-container {
            background: white;
            padding: 20px;
            border-radius: 8px;
            margin-bottom: 20px;
            box-shadow: 0 2px 4px rgba(0,0,0,0.1);
        }
        .table-container {
            background: white;
            padding: 20px;
            border-radius: 8px;
```

```
            box-shadow: 0 2px 4px rgba(0,0,0,0.1);
            overflow-x: auto;
        }
        table {
            width: 100%;
            border-collapse: collapse;
        }
        th, td {
            padding: 12px;
            text-align: left;
            border-bottom: 1px solid #ddd;
        }
        th {
            background-color: #f9f9f9;
            font-weight: bold;
        }
    </style>
</head>
<body>
    <div class="header">
        <h1 id="strategy-name">QTrader</h1>
        <p>
            <span id="mode"></span> |
            <span id="frequency"></span> |
            <span id="current-time"></span>
        </p>
    </div>

    <div class="metrics">
        <div class="metric-card">
            <h3>账户总资产</h3>
            <div class="value" 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>当前价格</th>
                    <th>市值</th>
                    <th>盈亏</th>
                </tr>
            </thead>
            <tbody></tbody>
        </table>
    </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: '#4CAF50',
                        tension: 0.1
                    },
                    {
                        label: '基准收益',
                        data: [],
                        borderColor: '#2196F3',
                        tension: 0.1
                    }
                ]
            },
            options: {
                responsive: true,
                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;
        document.getElementById('mode').textContent = data.mode;
        document.getElementById('frequency').textContent = data.frequency;
        document.getElementById('current-time').textContent = data.current_dt || '';

        // 更新指标
        if (data.portfolio) {
            document.getElementById('total-value').textContent =
                '¥' + data.portfolio.total_value.toLocaleString();

            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();
            }

            // 更新持仓表格
            if (data.positions) {
                const tbody = document.querySelector('#positions-table tbody');
                tbody.innerHTML = '';

                data.positions.forEach(pos => {
                    const row = tbody.insertRow();
                    row.insertCell(0).textContent = pos.symbol;
                    row.insertCell(1).textContent = pos.amount;
                    row.insertCell(2).textContent = '¥' + pos.price.toFixed(2);
                    row.insertCell(3).textContent = '¥' + pos.value.toLocaleString();

                    const pnlCell = row.insertCell(4);
                    pnlCell.textContent = '¥' + pos.pnl.toLocaleString();
                    pnlCell.className = pos.pnl >= 0 ? 'positive' : 'negative';
                });
            }
        }

        // 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));
        });

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

        // 初始化
        initChart();
    </script>
</body>
</html>
```

12. 分析与报告

12.1 PerformanceAnalyzer

```python
# analysis/performance.py

import pandas as pd
import numpy as np
from typing import Dict
from ..core.context import Context

class PerformanceAnalyzer:
    """绩效分析器"""

    def __init__(self, context: Context):
        self.context = context

    def calculate_metrics(self) -> Dict:
        """计算绩效指标"""
        portfolio_history = self._get_portfolio_history()

        if len(portfolio_history) == 0:
            return {}
```

```
        returns = self._calculate_returns(portfolio_history)
        benchmark_returns = self._get_benchmark_returns()

        metrics = {
            'total_return': self._total_return(returns),
            'annual_return': self._annual_return(returns),
            'sharpe_ratio': self._sharpe_ratio(returns),
            'max_drawdown': self._max_drawdown(portfolio_history),
            'win_rate': self._win_rate(),
            'benchmark_return': self._total_return(benchmark_returns),
            'alpha': self._calculate_alpha(returns, benchmark_returns),
        }

        return metrics

    def _get_portfolio_history(self) -> pd.DataFrame:
        """获取账户历史记录"""
        if not hasattr(self.context.portfolio, 'history'):
            return pd.DataFrame()

        return pd.DataFrame(self.context.portfolio.history)

    def _calculate_returns(self, portfolio_history: pd.DataFrame) -> pd.Series:
        """计算收益率序列"""
        values = portfolio_history['total_value']
        returns = values.pct_change().fillna(0)
        return returns

    def _get_benchmark_returns(self) -> pd.Series:
        """获取基准收益率序列"""
        benchmark_data = self.context.benchmark_manager.get_benchmark_data()
        df = pd.DataFrame(benchmark_data)

        if len(df) == 0:
            return pd.Series()

        values = df['value']
        returns = values.pct_change().fillna(0)
        return returns

    def _total_return(self, returns: pd.Series) -> float:
        """总收益率"""
        if len(returns) == 0:
            return 0.0
        return (1 + returns).prod() - 1

    def _annual_return(self, returns: pd.Series) -> float:
        """年化收益率"""
        days = len(returns)
        if days == 0:
            return 0.0
        total_return = self._total_return(returns)
        return (1 + total_return) ** (252 / days) - 1

    def _sharpe_ratio(self, returns: pd.Series, risk_free_rate: float = 0.03) -> float:
        """夏普比率"""
        if len(returns) == 0 or returns.std() == 0:
            return 0.0
        excess_return = returns.mean() * 252 - risk_free_rate
        return excess_return / (returns.std() * np.sqrt(252))

    def _max_drawdown(self, portfolio_history: pd.DataFrame) -> float:
        """最大回撤"""
        values = portfolio_history['total_value']
        cumulative = values
        running_max = cumulative.expanding().max()
        drawdown = (cumulative - running_max) / running_max
        return drawdown.min()

    def _win_rate(self) -> float:
        """胜率"""
        closed_trades = self._get_closed_trades()
        if len(closed_trades) == 0:
            return 0.0
```

```
        wins = sum(1 for t in closed_trades if t['pnl'] > 0)
        return wins / len(closed_trades)

    def _calculate_alpha(self, returns: pd.Series, benchmark_returns: pd.Series) -> float:
        """计算Alpha"""
        if len(returns) == 0 or len(benchmark_returns) == 0:
            return 0.0

        strategy_total = self._total_return(returns)
        benchmark_total = self._total_return(benchmark_returns)

        return strategy_total - benchmark_total

    def _get_closed_trades(self) -> list:
        """获取已平仓交易"""
        filled_orders = self.context.order_manager.get_filled_orders()

        trades = []
        buy_orders = {}

        for order in filled_orders:
            if order.side.value == 'buy':
                if order.symbol not in buy_orders:
                    buy_orders[order.symbol] = []
                buy_orders[order.symbol].append(order)
            else:
                if order.symbol in buy_orders and len(buy_orders[order.symbol]) > 0:
                    buy_order = buy_orders[order.symbol].pop(0)
                    pnl = (order.filled_price - buy_order.filled_price) * order.amount - order.commission - b
                    trades.append({
                        'symbol': order.symbol,
                        'pnl': pnl
                    })

        return trades
```

13. 实施路线图

13.1 阶段一：核心框架（2周）

- [x] Context、Engine、Scheduler
- [x] TimeManager
- [x] 数据工具接口定义
- [x] 订单、持仓、账户模型
- [x] 基础撮合引擎
- [ ] 测试日频回测

13.2 阶段二：撮合与基准（1周）

- [x] 完整撮合逻辑（限价单、涨跌停）
- [x] 滑点和手续费
- [x] BenchmarkManager
- [ ] 测试分钟频、Tick频

13.3 阶段三：数据工具示例（1周）

- [x] 基于Stock API的实现
- [x] 本地缓存优化
- [ ] 使用文档

13.4 阶段四：可视化系统（1-2周）

- [x] VisualizationServer设计
- [x] 前端模板
- [ ] WebSocket推送
- [ ] 测试回测与模拟盘

13.5 阶段五：状态管理（1周）

- [x] StateSerializer
- [ ] 自动保存点
- [ ] 恢复测试

13.6 阶段六：分析与报告（1周）

- [x] PerformanceAnalyzer
- [ ] ReportGenerator
- [ ] 完整报告

13.7 阶段七：文档与示例（1周）

- [ ] 用户手册
- [ ] API文档
- [ ] 策略示例

---

14. 总结

QTrader框架是一个专注于交易逻辑、数据完全解耦、支持实时可视化的A股量化交易系统。

核心特性

1. **数据解耦**：框架不自带数据系统，用户自由选择数据源
2. **精准撮合**：基于真实市场规则，限价单以档一价成交
3. **基准管理**：自动跟踪任意基准标的，每日更新收盘价
4. **实时可视化**：Web界面实时展示回测/模拟盘运行状态
5. **多频率支持**：日频、分钟频、Tick频（处理超时）
6. **回测/模拟统一**：同一策略代码无缝切换

使用流程

1. 实现数据工具（继承CalendarTool、PriceTool）
2. 编写策略（继承Strategy）
3. 配置参数（YAML）
4. 运行并实时监控（Web界面）
5. 查看报告

文档版本: 3.0.0
最后更新: 2025年10月
状态: 完整设计，待实施