QTrader 量化交易框架设计文档 (v7)

版本: 7.0.0
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**的量化交易框架。它的核心是为策略开发者提供一个稳定、高效、可扩展的运行环境，使其可以专注于策略本身的研发，而无需关心底层的交易执行、状态管理和时间推进等细节。

1.2 核心特性

- **数据完全解耦**: 框架通过 `AbstractDataProvider` 接口与数据源解耦，用户可以接入任何来源的数据（API、本地文件、数据库等）。
- **统一的回测与模拟**: 同一套策略代码，通过修改配置即可在历史回测和实时模拟盘之间无缝切换。
- **精确的撮合引擎**: 支持市价单和限价单，能够处理多空头寸，并模拟滑点和手续费。
- **灵活的时间调度**: 支持日线、分钟线等级别的事件频率，并允许用户自定义 `handle_bar` 的触发时间点。
- **清晰的生命周期**: 提供 `initialize`, `before_trading`, `handle_bar`, `after_trading`, `broker_settle`, `on_end` 等一系列钩子，方便策略逻辑的组织。
- **强大的状态管理**: 支持在运行过程中暂停、恢复，甚至从任意历史节点“分叉”(Fork)出一个新的回测，用于假设分析。
- **实时监控**: 内置Web服务器，可通过浏览器实时监控账户净值、持仓、订单和日志。

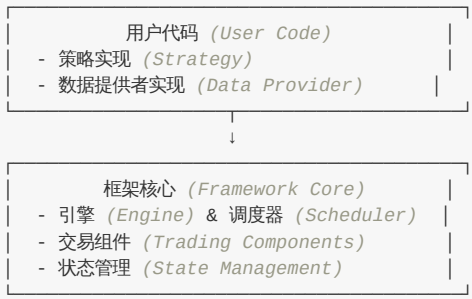
1.3 设计哲学

- **关注点分离 (Separation of Concerns)**: 框架、策略和数据三者分离。框架负责交易执行，策略负责交易决策，数据提供者负责提供行情。
- **依赖注入 (Dependency Injection)**: 框架不创建具体的数据提供者或策略实例，而是由用户在启动时注入，增强了灵活性和可测试性。
- **配置驱动 (Configuration Driven)**: 绝大多数行为，如运行模式、时间频率、交易成本等，都通过 `config.yaml` 文件进行配置，无需修改代码。

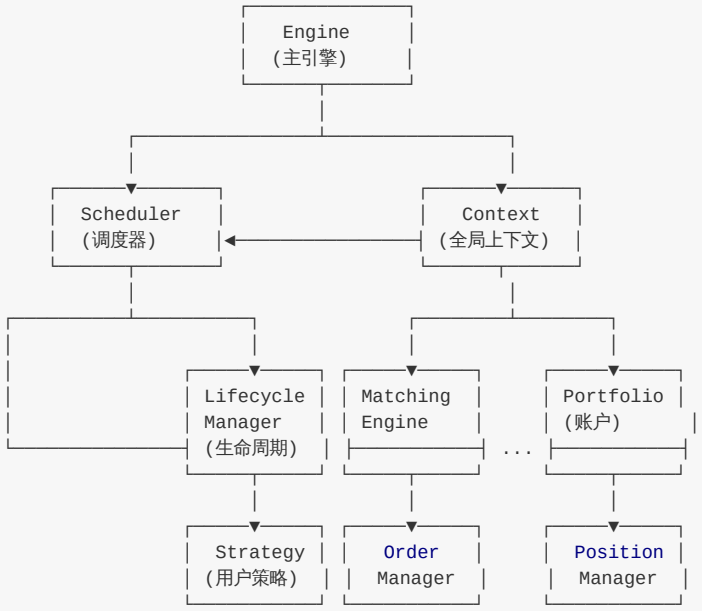
---

2. 系统架构

2.1 分层架构图



2.2 核心组件关系图



2.3 目录结构

```
qtrader/
├── core/          # 核心框架（引擎, 上下文, 调度器）
├── trading/       # 交易核心（账户, 订单, 持仓, 撮合）
├── data/          # 数据接口定义
├── strategy/      # 策略基类
├── runner/        # 运行器，提供高级API
├── analysis/      # 分析与可视化
├── utils/         # 工具模块（日志, 序列化）
└── configs/       # 配置文件与Schema
```

---

3. 核心引擎

核心引擎是 QTrader 的“中央处理器”，由一组紧密协作的组件构成，负责驱动整个事件循环、管理状态和执行策略。

3.1 引擎（`Engine`）：运行模式与生命周期

`Engine` 是框架的总协调官，负责管理一个完整的回测或模拟交易会话的生命周期。它通过 `run`，`resume`，`run_fork` 三个核心方法，提供了三种不同的运行模式，以满足从全新回测到复杂研究的各种需求。

1. 全新运行 (`run`)

这是最基础的模式，用于从零开始一个全新的回测或模拟交易。

- **工作流程 (`_run_backtest_new`)**:
  i. **加载用户代码**: 动态加载用户指定的 `strategy.py` 和 `data_provider.py` 文件。
  ii. **创建工作区**: 调用 `WorkspaceManager` 创建一个本次运行独有的工作目录，用于存放日志、代码/配置快照、状态文件和最终报告。
  iii. **初始化上下文 (`Context`)**: 创建一个全局 `Context` 实例，并注入配置文件。
  iv. **设置日志**: 将日志系统重定向到新创建的工作区内的日志文件。
  v. **初始化核心组件**: 依次创建 `Portfolio`，`OrderManager`，`PositionManager`，`BenchmarkManager`，`MatchingEngine`，`Scheduler` 等所有核心组件，并将它们注册到 `Context` 中。
  vi. **调用策略初始化**: `Scheduler` 调用策略的 `initialize()` 方法，此时策略可以访问已准备就绪的 `Context`。
  vii. **启动服务**: 如果配置中启用，则启动 `IntegratedServer` 用于Web实时监控。
  viii. **移交控制权**: 调用 `_execute_main_loop()`，将控制权移交给 `Scheduler` 的主事件循环，开始逐日/逐分钟的回测。
  ix. **最终处理 (`_finalize`)**: 循环结束后，执行收尾工作，包括保存最终状态、导出CSV数据、生成HTML报告等。

2. 恢复运行 (`resume`)

此模式用于从一个由“暂停”(Pause)操作生成的状态文件 (`.pkl`) 恢复一个中断的会话。

- **工作流程 (`_run_backtest_resume`)**:
  i. **加载状态**: 通过 `Engine.load_from_state()` 加载指定的 `.pkl` 文件。
  ii. **状态校验**: **严格检查**状态文件是否为可恢复的“暂停”状态。如果是由程序崩溃或正常结束时生成的终结状态文件，则会抛出异常，防止从一个不确定的状态恢复。
  iii. **恢复核心组件**: 从状态字典中反序列化并重建 `Portfolio`，`OrderManager`，`PositionManager`，`BenchmarkManager` 以及 `Context` 的核心属性（如 `current_dt`，`user_data` 等）。
  iv. **加载代码快照**: **强制加载**原始工作区中保存的 `snapshot_code.py` 和 `snapshot_data_provider.py`，以确保恢复的运行环境与暂停时完全一致。用户也可以选择性地覆盖数据提供者。
  v. **重新初始化非状态组件**: 重新创建 `Scheduler` 等无状态的组件。
  vi. **移交控制权**: 调用 `_execute_main_loop()`，并传入 `skip_initialize=True` 参数，确保**不会**再次调用策略的 `initialize()` 方法，然后从 `context.current_dt` 记录的时间点继续事件循环。

3. 分叉运行 (`run_fork`)

分叉是一种强大的“假设分析”(What-if Analysis)工具，它允许从一个历史状态文件“克隆”出一条新的时间线，保留历史状态，但应用新的逻辑继续运行。

- **工作流程 (`_run_from_snapshot`)**:
  i. **加载状态与校验**: 与恢复模式类似，加载并校验状态文件。
  ii. **加载新代码**: 加载用户**新指定**的 `strategy.py` （必须提供）和可选的 `data_provider.py`。
  iii. **创建全新工作区**: 为本次分叉运行创建一个全新的工作目录。
  iv. **精确重建分叉点状态**: 这是分叉模式最核心的步骤：
    - **截断历史**: 恢复 `Portfolio` 和 `BenchmarkManager`，但将其历史记录 (`history`) 截断到分叉日期之前。
    - **恢复持仓**: 从状态文件的**每日持仓快照** (`position_snapshots`) 中，找到分叉点**前一天**的收盘持仓，并以此作为新运行的初始持仓。
    - **恢复已成交订单**: 只恢复在分叉日期之前**已成交** (`FILLED`) 的订单历史。所有未成交订单被丢弃。
    - **处理策略状态 (`user_data`)**: 根据 `reinitialize` 参数决定是清空 `user_data` （默认）还是保留旧策略的状态。
  v. **调用策略初始化**: 如果 `reinitialize` 为 `True`，则调用新策略的 `initialize()` 方法。
  vi. **移交控制权**: 从分叉点的日期 (`fork_date_str`) 开始执行新的事件循环。

4. 模拟交易 (`simulation`)

模拟交易模式在 `run` 和 `resume` 流程中被统一处理（`_run_simulation_unified`），其核心是 `_synchronize_to_realtime` 时间同步机制。

- **时间同步机制**:
  i. **确定时间跨度**: 计算上次同步时间（对于新运行是当前时间，对于恢复则是状态文件中的时间）与当前真实墙上时钟之间错过了多少个交易日。
  ii. **清理瞬时状态**: 如果是恢复运行，将所有 `OPEN` 状态的挂单置为 `EXPIRED`，因为它们在真实世界中早已过期。
  iii. **快进结算**: 遍历所有错过的交易日，对每一天**只执行**简化的日终结算流程（`matching_engine.settle()`）。这会根据收盘价更新持仓市值和账户净值，但**不会**触发任何策略的生命周期钩子（如 `handle_bar`）。
  iv. **对齐当前时间**: 完成快进后，将 `context.current_dt` 设置为当前的真实时间，并判断当前市场处于盘前、盘中还是盘后，以便 `Scheduler` 从正确的状态开始实时事件循环。

3.2 上下文（`Context`）

`Context` 是一个全局上下文对象，作为框架内所有组件共享信息和状态的中央总线。它贯穿于整个交易的生命周期，并传递给策略的每一个生命周期钩子。

- **核心属性**:
  - **运行信息**: `mode`，`current_dt`，`frequency` 等。
  - **核心管理器**: 对 `portfolio`，`order_manager`，`position_manager` 等的引用。
  - **用户数据**: `user_data` 字典，用于策略内部的状态持久化。
  - **配置与服务**: `config`，`data_provider`，`logger` 等。

- **代码示例**:

```python
# 在策略中通过 context 访问核心功能
def handle_bar(self, context: Context):
    # 获取当前可用现金
    cash = context.portfolio.available_cash

    # 获取持仓
    position = context.position_manager.get_position("000001.XSHE")

    # 提交订单
    context.order_manager.submit_order("000001.XSHE", 100, "market")

    # 读写用户自定义数据
    ma_period = context.get('ma_period', 20)
    context.set('last_price', 10.5)
```

- **数据持久化规则**:
  `Context` 内不同数据的生命周期不同，尤其是在每日结算后：

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
账户现金（`portfolio.cash`）	✔ 是	账户资金状态会保留
持仓信息（`position_manager`）	✔ 是	所有持仓会保留到下一天
当日订单（`order_manager`）	✖ 否	当日未成交订单会被清空或标记为过期

3.3 调度器（`Scheduler`）：事件循环的设计

`Scheduler` 是框架的“心脏”，负责根据时间精确地生成和分派一系列生命周期事件。它为回测和模拟交易设计了两套不同的事件循环机制。

1. 回测事件循环 (`_run_backtest`)

回测模式是一个**确定性的、离线的**事件循环，严格按照预设的时间表推进。

- **核心逻辑**:
  i. **获取时间表**: 首先，通过 `TimeManager` 获取回测范围内的所有交易日。然后，根据配置的频率 (`daily` / `minute`) 和交易时段，构建出一天内所有 `handle_bar` 事件的精确触发时间点列表 (`_schedule_points`)。
  ii. **逐日推进**: 主循环遍历每一个交易日。
  iii. **每日事件序列**: 在每个交易日内部，严格按照以下顺序和预设时间触发事件：
    - `before_trading`
    - 循环触发所有 `handle_bar` 事件，并在每次之后调用 `matching_engine.match_orders()`
    - `after_trading`
    - `broker_settle` 和 `matching_engine.settle()`
  iv. **恢复逻辑**: 如果是从暂停状态恢复，循环会智能地跳到 `context.current_dt` 所在的日期，并只执行该日期中尚未完成的事件。

2. 模拟交易事件循环 (`_run_simulation`)

模拟交易是一个**实时的、在线的**事件循环，它基于真实世界的时钟运行一个**状态机**，以确保事件在正确的时间被触发。

- **核心逻辑**:
  i. **主循环**: 一个 `while` 循环，以大约1秒的间隔持续运行。
  ii. **每日状态重置**: 当真实日期进入新的一天时，重置状态机中的所有“每日标志位”（如 `before_trading_done`，`settle_done` 等）。
  iii. **市场阶段判断**: 在每一轮循环中，根据当前真实时间判断市场所处的阶段 (`BEFORE_TRADING`，`TRADING`，`AFTER_TRADING`，`SETTLEMENT`，`CLOSED`)。
  iv. **状态机驱动的事件分派**:
    - **盘前**: 如果当前是 `BEFORE_TRADING` 阶段且 `before_trading_done` 标志为 `False` ，则触发 `before_trading` 事件，并将标志位设为 `True` 。
    - **盘中 (`handle_bar`)**: 如果当前是 `TRADING` 阶段，调度器会检查当前时间是否已经越过了一个或多个预设的 `handle_bar` 时间点。
      - **不漏单机制**: 它会找到最后一个应该被执行但尚未执行的 `handle_bar` 时间点。
      - **容差处理**: 只有当真实时间与该 `handle_bar` 时间点的差距在一定容差范围内（例如分钟频为60秒），才会执行该 `handle_bar` ，以防止因系统卡顿而执行一个早已过期的Bar。过期的Bar会被跳过。
    - **盘后与结算**: 逻辑与盘前类似，根据市场阶段和对应的标志位来触发 `after_trading` 和 `broker_settle` 事件。
  v. **时间同步请求**: 如果 `LifecycleManager` 检测到策略代码阻塞超时，会设置 `context.resync_requested` 标志。 `Scheduler` 在事件循环中检测到此标志后，会立即调用 `Engine` 的 `_synchronize_to_realtime` 方法，强制进行时间校准，以确保系统的健壮性。

3.4 生命周期管理器 (`LifecycleManager`)：安全沙箱

`LifecycleManager` 的核心职责是为用户策略代码提供一个安全的执行环境（“沙箱”），确保用户代码的错误或不良性能不会导致整个框架崩溃或行为异常。这是通过两种关键机制实现的：

1. 异常隔离防火墙

所有对策略钩子方法（如 `initialize`，`handle_bar` ）的调用都被一个 `try...except` 块包裹。

- **机制**:
  - 当用户策略代码抛出任何未处理的异常时， `LifecycleManager` 会捕获该异常。
  - 记录详细的错误信息和堆栈追踪（Traceback）到日志中。
  - 将 `context.strategy_error_today` 标志位设为 `True` 。
  - **关键**: 捕获异常后，程序**不会**崩溃，而是会继续执行后续的框架逻辑（例如，进入下一个 `handle_bar` 或当天的结算流程）。这确保了单个策略的错误不会中断整个回测或模拟。

2. 阻塞检测看门狗 (仅限模拟交易)

在模拟交易模式下，策略代码的长时间阻塞是一个严重问题，因为它会导致框架的内部时钟与真实世界脱节。

- **机制**:
  - 在调用每个钩子方法之前，记录当前时间戳。
  - 在调用结束后，计算方法的执行耗时。
  - 如果耗时超过了配置中设定的阈值（`block_threshold_seconds`，默认为5秒），`LifecycleManager` 会：
  i. 在日志中打印一条**严重警告**，指出哪个钩子发生了阻塞以及阻塞了多长时间。
  ii. 将 `context.resync_requested` 标志位设为 `True`。
  - `Scheduler` 在其主循环中会检测到这个标志，并立即触发一次**时间同步**流程，强制将框架状态快进到当前的真实时间，从而修正因阻塞造成的时钟偏差。

---

4. 交易核心

交易核心是 QTrader 的执行层，负责将策略生成的交易意图（订单）转化为实际的账户和持仓变动。

4.1 账户模型 (`Portfolio`)

新版的 `Portfolio` 提供了更精细的会计科目，以精确支持多空策略。

- **核心指标**:
  - **`net_worth` (净资产)**: `现金 + 净持仓市值`，是衡量账户价值最核心的指标。
  - **`total_assets` (总资产)**: `现金 + 多头市值`。
  - **`long_positions_value` (多头市值)**: 所有多头持仓的当前市值总和。
  - **`short_positions_value` (空头市值)**: 所有空头持仓的当前市值总和（作为负债，以正数记录）。
  - **`margin` (保证金)**: 所有空头持仓占用的保证金总和。
  - **`available_cash` (可用现金)**: `现金 - 保证金`，是可用于开新仓的资金。

4.2 订单与持仓 (`Order` & `Position`)

- **`Order`**：表示一个交易订单，包含 `symbol`，`amount`，`side` (buy/sell)，`type` (market/limit) 等核心信息。
- **`Position`**：表示一个具体的持仓，其设计已升级以支持多空方向。
  - **`direction`**：`PositionDirection.LONG` 或 `PositionDirection.SHORT`。
  - **`available_amount`**：考虑 T+1 等规则后的可平仓数量。
  - **`today_open_amount`**：当日新开仓位数量，用于 T+1 计算。

4.3 撮合引擎 (`MatchingEngine`)

`MatchingEngine` 是模拟交易所的核心，负责处理订单的撮合与成交。为了保证回测的因果一致性（避免未来函数），撮合逻辑严格区分**即时订单**和**历史订单**。

- **价格确定规则**:

  - **市价单**:
  - 买入: 使用 `ask1` (卖一价)，若无则使用 `current_price`。
  - 卖出: 使用 `bid1` (买一价)，若无则使用 `current_price`。
  - **限价单 (即时订单)**:
  - 买入: 当 `limit_price >= ask1` 时，以 `ask1` 价格成交（而非限价）。
  - 卖出: 当 `limit_price <= bid1` 时，以 `bid1` 价格成交（而非限价）。
  - 若不满足条件，订单转为历史订单。
  - **限价单 (历史订单)**:
  - 买入: 当 `current_price <= limit_price` 时，以 `limit_price` 价格成交。
  - 卖出: 当 `current_price >= limit_price` 时，以 `limit_price` 价格成交。

- **订单状态转换**:

```
提交订单
    ↓
OPEN（即时订单）
    ↓
    ├─→ 当前Bar可成交 → FILLED
    │
    ├─→ 用户撤单 → CANCELLED
    │
    └─→ 当前Bar不可成交 → OPEN（转为历史订单）
            ↓
            ├─→ 后续Bar检查可成交 → FILLED
            │
            ├─→ 用户撤单 → CANCELLED
            │
            └─→ 当日收盘仍未成交 → EXPIRED
```

- **核心逻辑 (`process_trade`)**:
    - **买单 (`buy`)**:
        i. 优先用于**平掉**已有的**空头**仓位，并计算已实现盈亏。
        ii. 若买单数量在平空后仍有剩余，则用剩余数量**开立**或增加**多头**持仓。
    - **卖单 (`sell`)**:
        i. 优先用于**平掉**已有的**多头**仓位，并计算已实现盈亏。
        ii. 若卖单数量在平多后仍有剩余，并且系统允许做空，则用剩余数量**开立**或增加**空头**持仓。
- **风控检查**:
    - **市场规则**: 检查标的是否停牌、价格是否超出涨跌停限制。
    - **账户状态**: 检查购买是否有足够资金、卖出是否有足够可用持仓。
- **每日结算 (`settle`)**:
    - 在每个交易日结束时，执行结算流程，包括按收盘价更新持仓市值、记录每日净值、并根据 T+1 规则更新可用持仓。

---

5. 数据合约

数据合约是 QTrader 实现数据完全解耦的核心。它通过一个抽象基类 `AbstractDataProvider` 定义了框架运行所需的所有数据接口，强制用户提供一个符合规范的数据“适配器”。

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。其核心原则是：

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

5.2 `AbstractDataProvider` 接口定义

任何数据源（本地文件、数据库、实时 API 等）都必须通过实现这个接口来接入框架。

- `get_trading_calendar(start: str, end: str) -> List[str]`:
    - **职责**: 获取指定日期范围内的所有交易日。
    - **调用者**: `Scheduler` 在回测开始时调用，以确定时间循环的范围。

- `get_current_price(symbol: str, dt: datetime) -> Optional[Dict]`:
  - **职责**: 获取指定证券在特定时间点的价格快照。
  - **调用者**: `MatchingEngine` 在撮合和结算时频繁调用。
  - **返回**: 必须包含 `current_price`，可选 `ask1`, `bid1`, `high_limit`, `low_limit`。
- `get_symbol_info(symbol: str, date: str) -> Optional[Dict]`:
  - **职责**: 获取指定证券在特定日期的静态信息。
  - **调用者**: `MatchingEngine` 在交易前调用，用于风控检查。
  - **返回**: 必须包含 `symbol_name` 和 `is_suspended` (是否停牌)。
- **代码示例**:

```python
from qtrader.data.interface import AbstractDataProvider

class MyDataProvider(AbstractDataProvider):
    def get_trading_calendar(self, start: str, end: str) -> List[str]:
        # ... 实现从你的数据源获取交易日历的逻辑 ...
        return ['2023-01-03', '2023-01-04', ...]

    def get_current_price(self, symbol: str, dt: datetime) -> Optional[Dict]:
        # ... 实现获取价格快照的逻辑 ...
        return {
            'current_price': 10.5,
            'ask1': 10.51,
            'bid1': 10.49,
            'high_limit': 11.55,
            'low_limit': 9.45,
        }

    def get_symbol_info(self, symbol: str, date: str) -> Optional[Dict]:
        # ... 实现获取静态信息的逻辑 ...
        return {
            'symbol_name': '某某股票',
            'is_suspended': False,
        }
```

5.3 数据流

1. **启动时**: `Engine` 将用户实现的 `DataProvider` 实例注入到 `Context` 中。
2. **运行时**: `Scheduler`, `MatchingEngine` 等核心组件通过 `context.data_provider` 调用接口方法来获取所需数据。
3. **框架不关心**: 数据的具体来源、缓存策略或获取方式，这些完全由用户在自己的 `DataProvider` 实现中控制。

---

6. 策略

策略是用户交易思想的具体实现。在 QTrader 中，所有策略都必须继承自 `Strategy` 抽象基类，并实现其定义的生命周期钩子方法。

6.1 `Strategy` 基类与生命周期钩子

框架采用“好莱坞原则”（"Don't call us, we'll call you."），用户只需在正确的方法中定义好逻辑，`Scheduler` 会在正确的时间自动调用它们。

- `initialize(self, context: Context)` 【必须实现】
  - **时机**: 策略启动时调用一次。
  - **用途**: 初始化策略参数、设置股票池、预加载数据等。

- `before_trading(self, context: Context)` 【可选】
  - **时机**: 每个交易日开盘前。
  - **用途**: 更新股票池、计算当日信号、提交开盘前订单。
- `handle_bar(self, context: Context)` 【可选】
  - **时机**: 交易时段内根据配置的频率反复调用。
  - **用途**: 核心策略逻辑，根据行情数据生成并执行交易。
- `after_trading(self, context: Context)` 【可选】
  - **时机**: 每个交易日收盘后。
  - **用途**: 当日交易复盘、数据统计。
- `broker_settle(self, context: Context)` 【可选】
  - **时机**: 每日结算完成后。
  - **用途**: 与外部真实券商进行账户对账。
- `on_end(self, context: Context)` 【可选】
  - **时机**: 整个回测或模拟交易结束时。
  - **用途**: 全局性的数据分析、资源清理。

6.2 策略示例

以下是一个简单的双均线策略示例，展示了如何使用生命周期钩子和 `Context` 对象。

```
from qtrader.strategy.base import Strategy
from qtrader.core.context import Context

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

    def initialize(self, context: Context):
        """策略初始化"""
        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.XSHE')  # 交易标的
        context.set('price_history', [])
        context.logger.info("双均线策略初始化完成")

    def handle_bar(self, context: Context):
        """盘中策略执行"""
        symbol = context.get('symbol')

        # 1. 获取行情数据 (通过 data_provider)
        price_data = context.data_provider.get_current_price(symbol, context.current_dt)
        if not price_data:
            return
        current_price = price_data['current_price']

        # 2. 更新价格序列并计算指标
        price_history = context.get('price_history')
        price_history.append(current_price)
        if len(price_history) > context.get('ma_long'):
            price_history.pop(0)
        context.set('price_history', price_history)

        if len(price_history) < context.get('ma_long'):
            return

        ma_short = sum(price_history[-context.get('ma_short'):]) / context.get('ma_short')
        ma_long = sum(price_history) / context.get('ma_long')
```

```
        # 3. 生成交易信号并执行
        position = context.position_manager.get_position(symbol)

        # 金叉买入
        if ma_short > ma_long and not position:
            cash = context.portfolio.available_cash
            amount_to_buy = int(cash * 0.8 / current_price / 100) * 100  # 80%仓位
            if amount_to_buy > 0:
                context.order_manager.submit_order(symbol, amount_to_buy, "market")
                context.logger.info(f"金叉信号，买入 {amount_to_buy} 股 {symbol}")

        # 死叉卖出
        elif ma_short < ma_long and position:
            context.order_manager.submit_order(symbol, -position.total_amount, "market")
            context.logger.info(f"死叉信号，卖出 {position.total_amount} 股 {symbol}")
```

6.3 订单撤销策略示例

此示例展示了如何在特定时间挂一个限价单，并在收盘前如果订单未成交则将其撤销。

```
class CancellationStrategy(Strategy):
    def initialize(self, context: Context):
        context.set('symbol', '000001.XSHE')
        context.set('pending_order_id', None) # 用于存储待处理的订单ID
        # 在10:00和14:50触发handle_bar
        context.add_schedule("10:00:00")
        context.add_schedule("14:50:00")

    def handle_bar(self, context: Context):
        current_time = context.current_dt.time()
        symbol = context.get('symbol')

        # 10:00:00 - 挂限价单
        if str(current_time) == "10:00:00":
            price_data = context.data_provider.get_current_price(symbol, context.current_dt)
            if not price_data: return
            current_price = price_data['current_price']

            # 以低于当前价1%的价格挂一个买单
            limit_price = round(current_price * 0.99, 2)
            order_id = context.order_manager.submit_order(symbol, 100, "limit", price=limit_price)
            if order_id:
                context.set('pending_order_id', order_id)
                context.logger.info(f"已提交限价买单 {order_id} @ {limit_price:.2f}")

        # 14:50:00 - 检查并撤单
        if str(current_time) == "14:50:00":
            pending_order_id = context.get('pending_order_id')
            if pending_order_id:
                # 检查订单是否仍然是OPEN状态
                open_orders = context.order_manager.get_open_orders()
                is_still_open = any(o.id == pending_order_id for o in open_orders)

                if is_still_open:
                    context.logger.info(f"订单 {pending_order_id} 仍未成交，准备撤销。")
                    context.order_manager.cancel_order(pending_order_id)
                else:
                    context.logger.info(f"订单 {pending_order_id} 已成交或处理完毕。")

                context.set('pending_order_id', None) # 清理状态
```

7. 运行与配置

QTrader 的所有行为都由配置文件驱动，并通过一个高级的运行器（Runner）来启动。

7.1 运行器（`BacktestRunner`）

`BacktestRunner` 为 `Engine` 提供了一个简洁的程序化接口，是启动、恢复或分叉回测的标准入口。

- **`run_new(...)`**：启动一个全新的回测或模拟交易。
- **`run_resume(...)`**：从一个由“暂停”操作生成的状态文件恢复运行。
- **`run_fork(...)`**：从一个历史状态文件分叉出一个新的运行实例，用于假设分析。
- **代码示例（`run.py`）**:

```
from qtrader.runner.backtest_runner import BacktestRunner

if __name__ == '__main__':
    # 1. 指定配置文件路径
    config_path = 'config.yaml'

    # 2. 指定策略和数据提供者的代码路径
    strategy_path = 'strategy.py'
    data_provider_path = 'data_provider.py'

    # 3. 启动全新回测
    BacktestRunner.run_new(
        config_path=config_path,
        strategy_path=strategy_path,
        data_provider_path=data_provider_path
    )
```

7.2 配置文件（`config.yaml`）

`config.yaml` 是 QTrader 的“控制面板”，几乎所有的行为都可以在此配置。下面是一个更详尽的示例，包含了大部分常用配置项及其说明。

- **详尽示例（`config.yaml`）**:

```
# 引擎配置: 控制框架的整体运行模式和时间
engine:
  mode: backtest                # backtest (回测) 或 simulation (模拟盘)
  start_date: "2023-01-01"      # 回测开始日期
  end_date: "2023-12-31"        # 回测结束日期
  frequency: daily              # daily (日频) 或 minute (分钟频)

# 账户配置: 定义初始资金和交易规则
account:
  initial_cash: 1000000         # 初始资金
  trading_rule: 'T+1'           # 'T+1' 或 'T+0'
  trading_mode: 'long_only'     # 'long_only' (只做多) 或 'long_short' (多空)
  short_margin_rate: 0.2        # 开空仓所需的保证金率 (20%)

# 交易成本配置
matching:
  slippage:
    rate: 0.001                 # 固定滑点率 (0.1%)
  commission:
    buy_commission: 0.0002      # 买入佣金率 (万分之二)
    sell_commission: 0.0002     # 卖出佣金率
    sell_tax: 0.001             # 卖出印花税率 (千分之一)
    min_commission: 5.0         # 单笔最低佣金

# 生命周期钩子与交易时段配置
lifecycle:
  # 定义A股的常规交易时段
  trading_sessions:
```

```yaml
      - ["09:30:00", "11:30:00"]
      - ["13:00:00", "15:00:00"]
    # 定义各生命周期钩子的触发时间
    hooks:
      before_trading: "09:15:00"
      # handle_bar 支持单个或多个时间点（仅日频模式）
      # handle_bar: "14:55:00"
      handle_bar:
        - "10:00:00"
        - "14:55:00"
      after_trading: "15:05:00"
      broker_settle: "15:30:00"

# 基准配置
benchmark:
  symbol: "000300.XSHG"          # 用于对比的基准指数代码
  name: "沪深300"

# 服务器与报告配置
server:
  enable: true                   # 是否启用Web实时监控
  port: 8050
  auto_open_browser: true        # 启动时是否自动打开浏览器
report:
  enable: true                   # 是否在结束后生成HTML报告
  auto_open: true                # 是否自动打开报告

# 日志配置
logging:
  level: INFO                    # 日志级别: DEBUG, INFO, WARNING, ERROR
```

---

8. 分析与可视化系统

为了提供直观的回测分析和实时监控能力，QTrader 内置了一个集成的Web服务器（`IntegratedServer`），它负责数据收集、性能计算和前端展示。

8.1 技术架构

- **后端**: 基于 `Flask` 构建Web服务框架，处理HTTP请求；使用 `Flask-SocketIO` 实现与前端的实时双向通信。
- **前端**: 使用 `Jinja2` 模板引擎动态生成HTML页面，内置 `Chart.js` 用于绘制图表，并通过原生JavaScript处理SocketIO事件和DOM更新。
- **文件监控**: 使用 `watchdog` 库监控工作区文件的变化，实现数据的准实时更新。

8.2 核心功能

1. **实时监控仪表盘**:
    - **启动**: 在配置文件中将 `server.enable` 设置为 `true` 即可启动。
    - **数据流**: `Scheduler` 在每个关键事件点（如 `handle_bar` 之后）调用 `server.trigger_update()`。该方法会异步地从 `Context` 中收集最新的账户、持仓、订单、日志等数据，并通过SocketIO的 `update` 事件推送给所有连接的前端客户端。
    - **前端交互**: 前端页面通过 `/api/initial_data` 获取初始全量数据，然后监听 `update` 事件进行增量更新，实现了低延迟的实时监控。

2. **引擎远程控制**:
    - 服务器提供了一个 `/api/control` REST端点，允许前端页面通过发送POST请求来远程控制 `Engine` 的运行状态。
    - **支持指令**: `pause`, `resume`, `stop`。这使得用户可以直接在Web界面上暂停、恢复或停止一个正在进行的回测或模拟交易。

3. **静态HTML报告生成**:

- **触发**: 在一次运行正常结束后 (`_finalize` 阶段)， `Engine` 会调用 `server.generate_final_report()` 。
- **流程**: 该方法会一次性地从 `Context` 中收集完整的回测周期数据，计算最终的性能指标（通过 `PerformanceAnalyzer` ），然后使用 `Jinja2` 模板将所有数据渲染成一个独立的、可离线查看的 `report.html` 文件，并保存到当前的工作区目录中。

8.3 性能分析 (`PerformanceAnalyzer`)

`IntegratedServer` 在生成报告时，会依赖 `PerformanceAnalyzer` 来计算一系列专业的风险和性能指标。

- **数据来源**: 分析器直接从 `context.portfolio.history` (每日净值) 和 `context.order_manager.get_all_orders_history()` (历史成交订单) 中获取数据。
- **核心指标计算**:
  - **收益指标**: 年化收益率、累计收益率等。
  - **风险指标**: 年化波动率、最大回撤。
  - **风险调整后收益**: 夏普比率、卡玛比率。
  - **相对指标**: 阿尔法、贝塔（如果设置了基准）。
  - **交易分析**: 胜率、盈亏比、持仓周期等。
- **实现**: 底层大量使用了 `empyrical` 库来进行专业的金融指标计算。

---

9. 高级特性

除了核心的回测与模拟功能外，QTrader 还提供了一系列高级特性，以满足更复杂的研发和交易需求。

8.1 状态管理 (暂停/恢复/分叉)

QTrader 拥有强大的状态管理能力，允许用户在运行时与引擎进行深度交互。

- **暂停 (Pause)**: 在回测或模拟交易过程中，可以随时发出暂停指令。引擎会在处理完当前事件后安全地暂停，并将此刻的完整系统状态（包括账户、持仓、订单、策略内部变量等）序列化为一个 `.pkl` 文件。

- **恢复 (Resume)**: 可以从一个由“暂停”操作生成的状态文件无缝地恢复运行，继续执行后续的交易流程。

- **分叉 (Fork)**: 这是 QTrader 最具特色的功能之一。用户可以从任意一个历史状态文件“分叉”出一个全新的运行实例。这意味着可以保留该时间点之前的所有交易历史和持仓，但从该点开始，应用**新的策略逻辑**、**新的数据源**或**新的配置**继续运行。这对于进行“假设分析”（What-if Analysis）非常有用。

8.2 实时监控

当在配置文件中启用 `server` 选项后，QTrader 会启动一个内置的 Web 服务器。用户可以通过浏览器访问指定端口（默认为 `8050` ），打开一个实时监控仪表盘。

- **功能**:
  - **实时更新**: 通过 WebSocket 实时推送最新数据。
  - **核心指标**: 展示账户净值、收益率、基准收益率、持仓市值等。
  - **图表展示**: 绘制权益曲线和基准曲线。
  - **表格详情**: 显示当前持仓的详细信息（成本、数量、市值、盈亏等）和最近的订单记录。
  - **交互操作**: 提供“暂停”、“恢复”、“停止”等按钮，可以直接在网页上控制引擎的运行状态。

8.3 账户对齐

在模拟交易中，策略的内部状态可能会因为各种原因（如API延迟、网络问题、手动干预等）与外部的真实券商账户产生偏差。

QTrader 提供了 `align_account_state` 方法，允许用户在 `broker_settle` 钩子中，根据从外部获取的真实账户状态（现金和持仓），强制校准框架内部的 `Portfolio` 和 `PositionManager` ，确保两者在下一个交易日开始前保持一致。