Quant Trading bot design doc

Glossary

Problem

1. Human is emotional - we want to leverage machine based trading system to eliminate human footprint as much as possible.
2. We lived in a timezone that not in U.S. time. With the timezone difference we are hard to keep awake and focus when market open
3. We want to have high rewards on trading high risk financial products. Like crypto/future/杠杠index like TQQQ/TSLL. But we do not know how much risk we carry compare to the gain.
4. We are software engineers that have certain expertise on programing and AI, which want to leverage that. But we lack of the proper/fundamental knowledge about financial part. We are navigating in dark so we want to seek help to address this . So we need something expert on this alpha to educate us, instruct us what to do.

Schedule

3.30~4.6
Goal: Complete a runnable program and implement basic trading functionality before next week's meeting.

Requirements/constraints

一、自动交易机器人（Trading Bot）需求

- **自动化执行能力（Automation）**
    - 系统必须能够实时、自动地监听和解析金融Alert（来自Discord或类似平台）。
    - 自动将Alert转化为券商API所需的标准交易指令，并自动执行。

- **衡量指标：**
    - 自动交易Alert覆盖率 ≥ 95%（即不需人为干预直接交易成功的Alert比例）。
    - 从接收Alert到实际下单的端到端延迟 ≤ 500 ms。
- 【备注：考虑到你们的情绪问题，这里明确自动化覆盖率和延迟指标是关键的。】
- **风险管理（Risk Management）**
    - 系统需具备基本的风险管理功能，包括自动止损（Stop-loss）和止盈（Take-profit）。
    - 系统应能够识别并防止异常大额亏损或非正常交易行为（如交易金额超出预定限额）。
- **衡量指标：**
    - 单笔交易亏损控制在总账户资产的 ≤ 2%。
    - 系统能够准确识别并停止超过预定风险阈值的交易 ≥ 99%。
- 【备注：自动化系统必须有清晰的风险控制机制防止意外风险，尤其是高波动性品种。】

二、策略评估和组合管理机器人（Regression & Portfolio Bot）

- **策略回测与评估能力（Strategy Backtesting）**
    - 提供明确的历史回测功能，支持策略有效性的自动验证。
    - 系统可以使用过去至少1年的历史数据，对新策略进行回测和绩效评估。
- **衡量指标：**
    - 提供的回测指标包括但不限于：
        - **累积收益率（Cumulative Return）**【整体盈利能力】
        - **年化收益率（Annualized Return）**【投资回报年平均】
        - **最大回撤（Max Drawdown）**【投资期间最大的资产缩水幅度，越小越好】

        - **夏普比率（Sharpe Ratio）**【衡量收益和风险的比例，数值越高越佳，通常>1即良好】

- 【备注：这些指标是行业内标准，也是通俗易懂的衡量方式，你可以通过它们直观看到策略优劣。】

- **投资组合自动管理能力（Portfolio Management）**

    - 系统能够定期自动进行投资组合再平衡（Rebalancing），维持账户内资产配置在预定范围内。

- **衡量指标：**

    - 每月自动进行一次再平衡，确保实际资产组合偏离目标组合的误差 ≤ 5%。

    - 再平衡成功执行率 ≥ 95%（无需人工干预即可顺利完成）。

- 【备注：组合再平衡确保你们的投资一直处于你预期的风险收益平衡状态。】

三、策略组合与优化能力（Strategy Combination & Optimization）

- **外部策略和自定义策略的整合能力**

    - 支持自动接入外部策略（来自金融Alert）与用户自定义策略的结合，并能自动执行策略对比和优化。

- **衡量指标：**

    - 系统每月可支持至少1-3个外部策略接入与整合。

    - 自动对比新旧策略效果，输出对比报告成功率 ≥ 95%。

- 【备注：明确了外部策略的整合和自有策略对比，使系统可以验证哪种策略组合更有效。】

四、成本控制与性能要求（Cost & Performance）

- **成本效益控制（Cost Control）**

    - 系统每月的运行成本（包括服务器、券商API、数据订阅费用）应在团队设定的预算范围内。

- **衡量指标：**
    - 每月运行总成本 ≤ 设定的月度预算上限（如$500 USD）。
- 【备注：可在项目启动前明确设定这个预算，监控执行。】
- **性能要求（Performance）**
    - 系统需确保高效、低延迟运行，及时抓住市场交易机会。
- **衡量指标：**
    - 整个交易系统（从接收Alert到下单成交）的平均延迟 ≤ 500 ms，最大延迟不超过1秒

4.6 新增需求 （）

在近期美国总统唐纳德·特朗普宣布大规模关税措施后，全球金融市场经历了剧烈动荡，导致主要股指大幅下跌，引发了对全球经济衰退的担忧。

鉴于此类突发性的市场崩盘事件，建议在交易机器人中新增一个**紧急市场监控与响应模块（Emergency Market Monitor and Response Module）**，其主要功能包括：

1. **实时监控全球金融市场指标**：持续获取并分析主要股指（如S&P 500、道琼斯工业平均指数等）的实时数据，以及重要财经新闻，以检测异常波动或重大事件。
2. **设定预警阈值**：根据历史数据和风险偏好，设定市场波动的预警阈值。例如，当S&P 500指数在短时间内下跌超过某个百分比，或检测到特定关键词的财经新闻时，触发预警。
3. **自动触发紧急响应**：在预警触发时，立即暂停所有新的交易指令，并执行预设的风险控制措施，如清仓所有持仓，以防止进一步损失。
4. **高级别通知**：通过电子邮件、短信或其他即时通讯工具，向用户发送紧急通知，详细说明触发预警的原因和已执行的措施，确保用户及时了解情况。

通过引入该模块，交易机器人能够在市场出现剧烈波动时，自动采取保护性措施，降低潜在损失，并确保用户及时收到关键通知。

Functional requirements

明白了，我会继续围绕 Trade Bot 的核心功能闭环（监听 Discord Alert → 解析 → 下单）生成详细的 FUNCTIONAL REQUIREMENTS 实现方案。

我会在设计中预留未来扩展配置管理与交易日志可视化的结构，但不会在 P0 阶段实现这些功能，方便后续进入 P1 时快速接入。

稍等片刻，我整理好后马上提供给你。

交易机器人 P0 阶段功能实现设计

本设计聚焦于**P0 阶段**的交易机器人功能，实现从 Discord 获取交易警报到通过券商 API 自动下单的闭环流程。采用 Python 编写，运行于本地服务器（由用户手动在交易前启动，交易完成后关闭）。设计将支持**模拟交易模式**与**实盘交易模式**的切换，并以 **IBKR** 或 **MooMoo** 券商接口为优先实现对象。下文将详细说明模块划分、核心功能接口、模式切换、券商适配和数据结构设计，并标注 P0 阶段的 TODO 留口以便后续 P1 阶段扩展。

1. 模块结构划分

P0 阶段交易机器人的架构可以划分为以下主要模块，它们各司其职并串联形成完整的交易流程：

- **监听模块（Listener）**：负责连接 Discord 并监听特定频道的警报消息。一旦收到新的警报消息，就将其传递给解析模块处理。

- **解析模块（Parser）**：负责解析 Discord 警报消息的内容，将其中的**股票代码（symbol）**、**价格（price）**、**方向（bull/bear）**、**策略编号**、**市场数据**等信息提取出来，标准化封装为程序内部可用的数据结构。

- **风控模块（RiskGuard）**：负责对解析后的警报信息进行风险评估和策略校验。在下单前执行一系列检查（如交易信号有效性、仓位风险、资金充足性等），决定是否允许下单或对下单参数进行调整（例如设定下单数量、过滤过频繁信号等）。

- **券商执行模块（Broker Executor）**：负责与券商 API 交互下单。通过统一的**券商适配器接口**调用具体券商（如 IBKR、MooMoo、TradeStation）的 API，将交易指令发送到对应券商的交易系统。该模块封装实际 API 调用细节，实现下单动作的执行。

- **主控逻辑模块（Main Controller）**：作为各组件的调度中心，控制流程顺序。它初始化各模块，根据监听模块收到的警报依次调用解析、风控和券商执行模块，实现交易闭环。主控逻辑还负责根据配置选择运行模式（模拟或实盘）和目标券商，并在关键步骤加入日志记录等（P0 阶段简要记录，P1 阶段可扩展为完整日志/回放系统）。

上述模块以松耦合方式协作：监听模块检测到警报后触发解析模块，将得到的标准化数据传给风控模块检验，最后交由券商执行模块下单。各模块之间通过预定义的数据对象（如 **AlertInfo** 警报信息对象）进行信息传递。整个架构设计注重模块化和可扩展性，使得**模拟交易 vs 实盘**、**多券商适配**等在P0阶段即可支撑，并为后续功能（如日志回放、图形界面配置）预留接口。

2. 各模块核心功能和接口

下面详细说明各模块的核心功能点和对外接口设计，列出主要的函数和数据结构，便于开发时参考实现。

监听模块（Listener）

核心功能：
监听模块负责连接 Discord 服务器并实时监控指定频道的消息流。当目标频道出现新的交易警报消息时，监听模块应及时捕获该消息并将其内容传递给解析模块处理。P0 阶段可以采用简单可靠的实现，例如使用 Discord 提供的 API 或现有库（如`discord.py`）登录机器人账户，加入目标服务器并订阅警报频道的消息事件。监听过程中需要考虑基本的健壮性，例如掉线重连（P0 可以手动重启替代，P1 可加入自动重连机制）。

接口设计：

- `start_listening(channel_id, callback)`：启动监听指定频道的新消息。参数为频道ID和消息回调函数。当监听到新消息时，调用提供的`callback(message)`将消息内容传出（通常由主控逻辑提供的回调函数指向解析流程）。

- `stop_listening()`：停止监听 Discord 消息（在交易结束或程序退出时调用，用于资源清理）。
  (P0 实现可选，此功能在手动停止程序时自然结束，P1 可完善为更平滑的停止流程)

*实现建议：使用异步事件驱动模式。当 Discord 上有新消息时，触发on_message事件，在事件handler中调用解析模块。监听模块本身不做业务逻辑判断，只负责把**原始消息字符串**传递出去。*

解析模块（Parser）

核心功能：
解析模块接收来自监听模块的原始警报文本（可能包含多行或特定格式内容），负责**提取关键交易要素**并封装为标准数据对象。例如，从警报消息中解析出：交易标的代码（如股票符号或合约

代号）、警报触发价格、信号方向（看多Bullish/看空Bearish）、策略编号或名称、相关的市场数据等字段。由于用户提供的警报格式固定且包含上述信息，解析模块可以基于已知格式实现高效解析。

具体来说，可使用字符串分析或正则表达式提取字段。例如，假设警报消息格式类似：

Bullish Bias
Detected Symbol: NQ
Price: 19656.00
Strategy: OrderFlowBot3.5 (ID 3)
Market: NDX 19455.68, SPX 4561.37 ... （市场相关数据）

解析模块应能**适配多语言格式**（若存在中英文版本警报）。例如，如果有中文警报格式，对应字段标识可能是**“看多”/“看空”**或其他术语，因此解析器需要针对不同语言/格式提供解析函数或配置。在 P0 阶段，可以假设警报格式固定不变（例如始终英文），实现单一解析逻辑；但设计时可考虑扩展性，如通过配置文件或策略模式为不同格式定制解析器。

接口设计：

- `parse_alert(message_str) -> AlertInfo`：解析给定的警报消息字符串，返回标准化的警报信息对象（例如 `AlertInfo` 类实例，详见第5节）。如果解析失败（格式不符或缺少字段），可返回 `None` 或抛出异常并由主控逻辑捕获处理。

- （可选）`detect_language(message_str) -> str`：如果需要支持多语言格式，可实现一个简单的语言/格式检测，根据消息内容选择相应解析逻辑。【P0 可以不实现，假定单一格式；P1 若接入多来源警报，可拓展此功能】。

实现建议：使用 Python 正则表达式或字符串操作提取字段。例如，可定义固定的正则模式匹配 "Symbol:\s(?P\w+)" 等来抓取值。解析完成后将各字段填充到 `AlertInfo` 数据结构实例中供后续模块使用。

风控模块（RiskGuard）

核心功能：
风控模块在交易执行前充当安全网，负责根据预先设定的规则对交易信号进行审查和调整，以控制风险。核心功能包括：

- **信号校验**：检查解析后的警报信息是否满足下单条件。例如，确认警报的`symbol`在支持的交易列表内、价格数值合理、方向明确等。如果警报信息有异常（如无法识别的代码，价格为空等），可以拒绝该交易信号。

- **仓位风险检查**：根据当前账户情况和预设风险参数决定是否可开新仓位。例如，检查账户余额/保证金是否足够执行该笔交易；如果是看空信号，判断账户是否允许做空该股票（有

无融券权限）等。P0 阶段可以简化为：每次信号都假定可以交易，但仍至少检查**是否重复持仓**（避免对同一标的发出多次相同方向订单）等基本规则。

- **下单参数调整**：确定实际下单的数量、价格和订单类型等。解析模块提供了标的符号和触发价，风控模块可以根据策略和风险偏好决定采用**市价单**还是**限价单**、购买数量多少等。P0 阶段若未集成复杂仓位管理，可采取简单策略：例如基于配置固定每次购买`N`股或根据预设的金额计算股数。在模拟交易模式下，可以更灵活地调试数量而不实际交易。

- **信号节流/过滤**：防止过于频繁的交易或重复信号。比如设定如果短时间内多次收到同一策略对同一标的的警报，仅执行第一次，后续忽略（或加入冷却时间）。此功能P0可选实现，P1可加强。

接口设计：

- `evaluate_alert(alert: AlertInfo) -> OrderInfo or None`：对解析后的警报信息进行评估，返回具体下单指令数据（例如 `OrderInfo`，包含最终决定的下单方向、数量、价格等）。如果风控检查未通过，则返回 None（表示不执行交易）。

    - 实现上，如果警报被拒绝，可在日志中记录原因（例如“RiskGuard拒绝下单：余额不足”）。

    - 若通过，则生成下单所需的结构。P0 阶段可简单地将 AlertInfo 的 symbol 和方向填入 OrderInfo，数量使用预设值，价格使用市价（price 字段可仅用于参考或限价单价）。

- （可选）`set_position_limits(...)` / `set_risk_params(...)`：设置风控相关的参数接口，如单笔交易最大金额、单标的允许的最大仓位等。【P0 可写死在配置，P1 提供配置接口】。

实现建议：P0 可将风控逻辑简化，但代码结构上为将来扩展预留。比如，可以在 `evaluate_alert` 内部先后调用多个检查函数（每个函数负责一种风险检查），这样在P1阶段可以很方便地增加/修改检查逻辑。还可以在 OrderInfo 中增加字段用于后续扩展（如止损价、目标价等），当前先不赋值，只作为占位。

券商执行模块（Broker Executor）

核心功能：
券商执行模块通过调用券商提供的 API 将交易指令发送到市场。由于需要支持多个券商并方便扩展，我们设计一个**券商适配器接口（BrokerAdapter）**来抽象下单操作。核心功能包括：

- **券商 API 连接**：与指定券商建立连接或会话。例如，对于 IBKR，需要启动 TWS 或 IB Gateway 并通过其 API 接口进行通信；对于 MooMoo（富途），需要调用其开放 API 接

口并通过本地 OpenD 服务交易。P0 阶段可以假定用户已经在本地运行好必要的券商客户端（如 IB Gateway），券商执行模块负责准备 API 调用所需的连接参数（如主机、端口、API密钥等）。

- **订单下达**：将风控模块传来的 OrderInfo 转换为券商 API 所需的请求格式，并调用相应方法下单。包括指定交易标的、买卖方向（**多头买入**或**卖出/做空**）、数量、订单类型（市价/限价）等信息。券商执行模块应统一这些操作接口，不同券商的具体实现细节将由各自的适配器类完成。

- **结果处理**：获取券商 API 的下单结果。例如，订单是否成功提交、返回的订单ID、若失败包含错误信息等。P0 阶段可以简单地判断调用是否报错，记录成功或失败；P1 阶段可扩展为详细的订单状态跟踪（已成交、部分成交、已取消等）。

- **模式切换支持**：券商执行模块也需要支持**实盘**和**模拟**模式，在模拟模式下不真的发送订单给券商，而是模拟一个成功响应（这一部分在第3节详细介绍）。

接口设计：
为实现多券商支持，定义一个抽象的券商适配器基类，例如:

```
class BrokerAdapter:
    def connect(self): ...
    def place_order(self, order: OrderInfo) -> bool: ...
    def disconnect(self): ...
```

每个具体券商有各自的实现类，实现上述接口：

- `IBKRBrokerAdapter(BrokerAdapter)`：通过 IBKR API 下单。实现时可能使用 IB 自带的 Python API 库或`ib_insync`等三方库。`connect()`负责连接到IB Gateway/TWS，【P0阶段】可以在初始化时自动连接一次。`place_order()`将 `OrderInfo` 映射为 IBKR 的下单请求，例如使用 `placeOrder(contract, order)` 方法提交。

- `MooMooBrokerAdapter(BrokerAdapter)`：通过 MooMoo(Futu) API 下单。例如使用富途开放平台的 API，将订单发送到相应市场。连接可能涉及 OpenD 的本地服务接口。

- `TradeStationBrokerAdapter(BrokerAdapter)`：通过 TradeStation API 下单。可能涉及 REST API 调用，使用HTTP请求发送订单。

- `MockBrokerAdapter(BrokerAdapter)`：模拟券商，用于**模拟交易模式**或测试。`place_order()`不实际发送交易，而是打印或记录下单信息，直接返回成功。

典型接口函数：

- `connect()`：建立与券商的会话连接。**返回**连接是否成功。对需要预连接的券商（如IBKR）调用其初始化流程；对于无需长连接的REST型券商，可以留空实现。P0简单实现即可，如 IBKR在对象创建时连接。

- `place_order(order: OrderInfo) -> bool`：执行下单操作。参数为标准化的订单信息，返回布尔值表示提交是否成功（或抛出异常表示失败）。在实现中，将 OrderInfo 映射到券商API字段，如符号、方向、数量、价格等，然后调用券商SDK。对于 `MockBrokerAdapter`，可以直接打印订单模拟执行。

- `disconnect()`：断开与券商的连接/会话。在程序结束时调用，确保资源释放。

实现建议： P0 阶段可以先实现 **IBKRBrokerAdapter** 和一个 **MockBrokerAdapter**（或使用 IBKR 的沙盒模式）满足基本需求。实现时注意：IBKR 等API调用是异步的，可采用同步包装或回调等待成交回报。由于P0不要求完整回报处理，可在下单后短暂停留确认提交成功即可。整个券商执行模块通过统一Adapter接口，可以在配置中选择使用何种券商实现，方便扩展新的券商而无需修改其他模块逻辑。

主控逻辑模块（Main Controller）

核心功能：
主控逻辑模块统筹整个交易机器人的运行流程。其职责包括：根据配置初始化各子模块（监听、解析、风控、执行），设定运行模式（模拟/实盘）和券商类型，然后进入监听->处理->下单的循环。在收到每条警报后，按顺序调用解析模块获取结构化数据，再调用风控模块评估生成订单，最后调用券商执行模块下单。主控模块还负责整体错误处理和简单日志记录，确保即使某一步失败也能继续监听后续信号或者安全退出。

接口设计：

- `run()`：主入口函数，启动交易机器人。执行步骤：

    1. **初始化配置**：读取用户配置（如使用的券商类型、运行模式、固定下单数量等）。初始化对应的 BrokerAdapter 实例，例如根据配置选择 IBKR 或 MooMoo，以及模拟还是实盘模式（详见第3节）。

    2. **启动监听**：调用 Listener 的 `start_listening(channel, callback)`，将回调设置为内部处理函数。例如：当有新消息时，调用 `handle_message(message)`。

    3. **处理回调**：`handle_message(message_str)` 内部依次执行：
        a. 调用 `Parser.parse_alert(message_str)` 得到 `AlertInfo` 对象。
        b. 若解析失败（返回 None 或异常），记录警告日志并跳过该消息。

c. 若解析成功，调用 `RiskGuard.evaluate_alert(alert_info)` 得到 `OrderInfo`。
d. 如果 RiskGuard 返回 None，表示此信号被风控拦截，记录原因并结束该消息处理。
e. 若得到有效的 `OrderInfo`，调用选定的 `BrokerAdapter.place_order(order_info)` 执行下单。
f. 根据返回结果，记录**交易执行日志**：成功则记录订单ID或"下单成功"，失败则记录错误信息。

4. **保持运行**：`run()` 调用后应保持进程不退出，持续监听频道消息（Discord监听自身通常是异步阻塞模式，主控逻辑可依赖它阻塞）。由于P0阶段在每次交易后人工关闭程序，可以不实现复杂的停止条件；当用户手动终止时再进行清理。

5. **清理退出**：当用户停止机器人时，调用 Listener 的 `stop_listening()` 和 BrokerAdapter 的 `disconnect()` 进行资源清理，安全退出程序。

- （可选）`configure(mode, broker, params)`：设置运行参数的接口。P0 可简单地通过读取配置文件或全局变量，不需要专门的配置函数。但在设计上预留此接口，P1 阶段可以通过它或一个配置模块来调整运行时参数（如切换券商或修改风控参数）。

实现建议： 主控逻辑应尽量简单明了，按顺序调用各模块功能。可以使用日志系统打印每步结果（P0 用简单打印或写文件，P1 可接入更完善的日志模块）。对于异常情况，用 `try/except` 捕获，例如券商API异常，捕获后记录并继续运行监听。主控模块也可以考虑在子模块之间传递上下文信息（如在 AlertInfo 或 OrderInfo 增加traceId用于日志关联等），以便后续调试和回放。在P0阶段这些都可留作注释或 TODO。

3. 模拟交易模式 vs 实盘交易模式设计

交易机器人需要支持**"纸上交易"（Paper Trading，模拟）**和**实盘交易**两种模式，并且做到以最小代价进行切换。这意味着大部分代码逻辑对两种模式应当通用，仅在关键接口处有所区别。设计思路如下：

- **配置驱动的模式切换：** 在系统配置中增加一个运行模式标志，例如 `TRADE_MODE = "paper"` 或 `"live"`。主控逻辑在初始化时读取该配置，决定采用模拟模式还是实盘模式。开发时可以通过环境变量、配置文件或命令行参数来设置此标志，实现"一处修改、全局生效"。

策略模式封装差异： 使用面向对象的**策略设计模式**来封装交易模式的差异。在本设计中，主要体现在券商执行模块：通过提供真实券商适配器（Real）和模拟适配器（Mock）两套实现，实现接口一致、内部行为不同的下单操作。主控逻辑根据运行模式选择相应的适配器实例。例如:

```
if CONFIG["MODE"] == "paper":
    broker = MockBrokerAdapter()
else:
    broker = IBKRBrokerAdapter()  # 或 MooMooBrokerAdapter 等
```

- 这样，后续的 `broker.place_order(order)` 调用对于主控来说是透明的：在实盘模式下真正发送交易，在模拟模式下由 Mock 实现内部模拟（比如打印模拟交易）。这种模式下，**除了一处创建BrokerAdapter的代码外，其余流程无需修改**，满足最小代价切换要求。

- **模拟模式的实现细节：**

    - **使用券商模拟环境 vs 自行模拟：** 一些券商本身提供模拟交易环境（如 IBKR 提供 paper account，MooMoo 也有模拟盘），如果使用这些环境，那么对于代码而言几乎无需区别处理——连接不同的模拟账户即可，下单接口相同。这也可视为一种“实盘模式”，只不过资金是虚拟的。因此，另一种模拟实现是在 BrokerAdapter 层面不实际调用券商API，而是模拟行为。两种方案各有优点：利用券商模拟盘可以测试完整流程但需要真实连接，使用 MockAdapter 则不依赖外部系统、方便快速测试逻辑。**P0 阶段**可任选一种或同时支持：例如，可以让 MockAdapter 完成简单日志记录，而如果用户希望更真实测试，也可配置使用 IBKR 的paper账户通过 IBKRBrokerAdapter 达到模拟目的。

    - **订单处理**：在模拟模式下，BrokerAdapter 的 `place_order` 可立即返回成功，并生成一个虚拟订单ID或者简单打印“[Paper] 下单 XXX股票 数量Y 成功”。也可以维护一张表记录所有模拟订单（P1 阶段用于回放或分析）。

    - **风险控制**：一般实盘和模拟模式下风控逻辑应保持一致，以确保模拟结果具有参考价值。设计上不建议为两种模式采用不同的风控规则，除非出于测试目的可以通过配置调整某些检查项开关。P0 阶段可统一处理。

- **模式切换验证：** 确保当切换配置后，程序能够正确调用对应模式的实现。例如，在测试中运行一次模拟模式，检查不会触发真实交易，再切换实盘模式，检查能够与真实券商通信。通过良好的抽象，开发者只需专注于各模式下模块本身的正确性，而无需改动整体流程代码。

总之，通过**配置+策略模式**，交易机器人可以**以最小代价**支持模拟与实盘两种运行模式：绝大部分模块（监听、解析、风控、主控）完全复用，仅券商执行模块通过适配器实现行为差异。这样的设计也方便了后续功能测试，开发者可以先在模拟模式下验证逻辑，再无缝切换到实盘执行真实交易。

4. 多券商适配架构设计

为了支持 IBKR、MooMoo、TradeStation 等不同券商接口，并方便未来扩展新的券商，系统采用**适配器设计模式**抽象券商接口。核心思想是定义统一的 **BrokerAdapter 接口**，不同券商实现该接口。在主控逻辑中使用接口编程而非依赖具体实现，从而实现**可插拔的券商支持**。设计要点如下：

- **统一的券商接口定义：** 在第2节券商执行模块部分，我们已大致定义了 `BrokerAdapter` 基类，其主要方法包括 `connect()`、`place_order(order)`、`disconnect()` 等。根据需要，还可以扩充例如查询账户余额`get_balance()`、查询持仓`get_positions()`等接口（P0 未必需要，但设计时可考虑）。关键是这些接口应当覆盖交易机器人的需要且能被不同券商实现。接口的函数签名和参数尽量设计得通用，比如 `place_order` 使用我们内部定义的 OrderInfo对象而非某券商特有格式。

- **券商实现适配器：** 针对每个支持的券商，实现一个适配器类继承自 BrokerAdapter，并封装其API调用。以 IBKR 为例：IBKR 需要先连接TWS/IB Gateway，然后下单需要构造 Contract 和 Order 对象。我们的 `IBKRBrokerAdapter.place_order()` 实现内部会根据 OrderInfo 创建 IBKR API需要的对象，再调用 IBKR 的 API 方法下单 (Placing Orders using TWS Python API | Trading Lesson) (Using Python, IBPy and the Interactive Brokers API to Automate Trades)。MooMoo 则需要通过 its Futu API 提交交易请求，可以在 `MooMooBrokerAdapter` 中调用 Futu API 的下单函数。TradeStation 提供 HTTP/REST API，则对应适配器会发起HTTP请求。**所有这些差异都被隐藏在各自适配器内部实现中**，对外表现一致。

适配器工厂/选择机制： 主控逻辑在运行开始时，根据配置的券商类型选择实例化对应的 BrokerAdapter 实现。例如配置文件中设置 `BROKER = "IBKR"` 或 `"MooMoo"`，主控逻辑会：

```
 broker_type = CONFIG["BROKER"]
if broker_type == "IBKR":
    broker = IBKRBrokerAdapter(...)
elif broker_type == "MooMoo":
    broker = MooMooBrokerAdapter(...)
# ...
```

- 可以将此选择过程封装到一个**工厂函数**或**工厂类**中，如 `BrokerAdapterFactory.create(broker_name)` 返回正确的实例，以保持主流程简洁。同样，这里也可以结合第3节的模式：若模式是paper且某券商无内置模拟盘，则返回 `MockBrokerAdapter` 实例。比如：如果配置 `BROKER="MooMoo"` 且 `MODE="paper"`，工厂可直接返回一个 `MockBrokerAdapter`，从而无需依赖真实富途环境。

- **多券商共存与扩展：** 在设计上确保可以同时支持多个券商实例（例如用户可能有多个账户希望同时下单）。P0阶段不需要实现多账号并行，但结构上不妨考虑，比如 BrokerAdapter实例不设计为单例，而是可根据需要创建多个。对于扩展新券商，只需新增一个适配器类，实现接口并在工厂方法中加入对应分支，不会影响其他模块。这种解耦

保证了当用户将来需要支持其他平台（如 Alpaca、Robinhood 等）时，可以快速集成。

- **处理券商差异：** 由于券商 API 在性能、风控侧有差异，适配器需要做相应处理。例如，有的券商下单接口是同步的，有的是异步的；有的需要先查询合约ID，有的直接使用符号。可以在适配器内实现一些辅助方法来弥合差异，确保 `place_order` 对上层来说尽可能是即时返回。在极端情况下，如果某券商流程与假设的接口差异过大，可以对 BrokerAdapter 接口做调整或在特定适配器中override额外流程，但总体思想是**不改变**上层模块与 BrokerAdapter 交互的接口契约。

通过以上设计，实现一个**高内聚、低耦合**的多券商适配层。对于主控、风控等模块来说，无论对接哪一家券商，都通过相同的方法调用下单；对于开发者来说，新增券商支持只需关注新适配器模块的实现，不影响系统其他部分。这满足了扩展性的要求，也方便调试 —— 可以用 MockAdapter 进行测试，再切换到真实券商适配器做实盘交易。

5. 模块间数据结构传递设计

各模块之间传递信息需要清晰的**数据结构**或对象来承载。良好的数据结构设计能够降低模块耦合、提高可读性。P0 阶段建议定义以下主要数据结构：

警报信息结构（AlertInfo）： 表示从 Discord 警报解析得到的标准化交易信号数据。可以使用 Python 的类或命名元组/dataclass来实现。例如定义一个 dataclass：

```
from dataclasses import dataclass
@dataclass
class AlertInfo:
    symbol: str       # 交易标的代码
    price: float      # 警报给出的价格（触发价或参考价）
    direction: str    # 方向: "bull" 或 "bear"（也可用Enum表示）
    strategy_id: str  # 策略编号或名称
    market_data: dict # 其他市场数据，如指数值等（可选，用dict或自定义类型）
    timestamp: datetime # 警报时间戳（可选，记录信号时间）
```

- Parser模块解析后会生成一个AlertInfo实例，包含了交易决策所需的关键信息。之所以用类而非直接用dict，是为了使字段含义明确，便于后续模块使用和扩展（例如增加字段）。当然，用 `dict` 也可以实现，但类结构有利于IDE提示和类型检查。**P0 阶段**可根据实际警报内容调整字段，例如如果警报未提供timestamp可以暂不使用该字段。`market_data`字段用于存放诸如“大盘指数”、“板块涨跌”等警报提供的附加信息（例如上例中的 SPX/NDX值），便于风控或决策时参考。

订单信息结构（OrderInfo）： 表示送往券商执行模块的交易订单参数。RiskGuard模块输出该结构，BrokerAdapter输入该结构。OrderInfo可以从AlertInfo派生，加上与下单相关的细节，例如实际下单数量、订单类型等。可定义如下：

```
@dataclass
class OrderInfo:
    symbol: str        # 交易标的代码
    action: str        # 动作: "BUY" 或 "SELL"（或 "SELL_SHORT" 区分做空）
    quantity: int      # 下单数量
    order_type: str    # 订单类型: "MKT"市价单 或 "LMT"限价单 等
    price: float = None # 下单价格(限价单需要), 市价单可为None或忽略
    strategy_id: str = None  # 来源策略编号（可选附加信息）
```

- 在P0阶段，可以采取简单约定：对于`bull`信号，`action`填 "BUY"；对于`bear`信号，填 "SELL"（如果采用空仓做空策略，则用 "SELL_SHORT" 明确做空动作）。`quantity`可以由RiskGuard根据预设决定，例如统一配置为100股或1张合约等。`order_type`可默认为 "MKT"（市价单）以确保立即成交；如果需要也可以支持限价单，例如使用警报给出的价格作为参考。OrderInfo还可以包含`stop_loss`、`take_profit`等扩展字段（P1阶段可能用到），P0先不使用但结构上可以留空以备扩展。

- **流程中的数据传递：**

    - **Listener -> Parser：** 监听模块获取到**原始消息字符串**，传递给解析模块。这里可以简单用字符串，因为解析模块接口就是接受字符串。

    - **Parser -> RiskGuard：** 解析后得到 AlertInfo 对象，传递给风控模块。可以直接调用 `riskguard.evaluate_alert(alert_info)` 将对象传递。由于 AlertInfo 是自定义类型，RiskGuard 可以直接访问其属性，如 `alert_info.symbol` 等。

    - **RiskGuard -> BrokerExecutor：** 风控模块返回 OrderInfo 对象（或返回None表示不下单）。主控逻辑拿到 OrderInfo 后，调用券商适配器 `place_order(order_info)`。券商模块内部再从 OrderInfo 提取必要字段调用外部API。

    - **BrokerExecutor -> 主控逻辑：** 券商适配器返回下单结果状态（成功/失败），主控逻辑基于此记录日志或做简单处理。在需要时，也可以将订单回执封装为一个结果对象，如 `OrderResult` 包含订单ID、状态等，但 P0 简单用 bool 或异常处理即可。

- **错误信息传递：** 在模块间传递过程中，如果发生错误（如解析错误、下单异常），应通过返回值或异常机制告知主控逻辑。P0 阶段，可在解析失败时返回 None，让主控逻辑判断处理；在下单失败时通过捕获异常或者返回 False 标记失败。可以定义统一的异常类，如 `TradingBotError`、`ParseError`、`OrderError` 等，提高代码可读性。**但注意**模块之间不应直接以异常作为正常流程一部分，除非真的是异常情况。大部分情况下，返回

None 或特定状态码由上层决定如何处理是更清晰的方式。

综上，建议使用**类或命名元组**来管理数据传递，使各模块接口清晰。上例的 AlertInfo 和 OrderInfo 为核心数据结构，开发者可根据实际警报信息补充字段。在P0实现时，可先确保基本字段正确传递，并通过简单打印或日志验证每步得到的数据结构内容是否符合预期。这种结构化的数据传递为后续功能扩展（例如在AlertInfo中加入更多维度信息、在OrderInfo中加入风控批注等）提供了便利。

6. P0 阶段的 TODO 留口和未来扩展

在 P0 阶段，我们专注于实现基本的警报监听和自动下单功能，舍弃了一些附加的复杂功能。但在设计中需为这些未来可能加入的功能预留接口或注释标记，以便 P1 阶段及以后能够平滑扩展。以下列出 P0 阶段明确的 TODO 留口和扩展点：

- **配置管理模块：** *（P1计划）* 当前配置（如券商类型、API密钥、交易模式、默认下单数量等）在P0中可能以全局变量或简单的配置文件读取实现。未来可引入专门的配置管理模块，支持从文件或GUI读取配置、热更新参数等。建议在代码中关键的配置点添加`# TODO: 将硬编码配置改为读取自配置模块`注释。

- **日志记录与回放：** *（P1计划）* P0 阶段日志通过简单打印或写文本文件实现，仅记录关键事件。未来需要更健全的日志系统，记录**每一笔警报、决策和下单结果**。例如，将 AlertInfo 和 OrderInfo 序列化存储，以支持**日志回放**功能——即日后可以读取日志重现某次警报触发时机器人的决策过程，用于调试或策略分析。设计上可在主控逻辑的每个步骤加入钩子，将数据发送给日志模块（P0 可留空或简单实现）。代码中可标记`# TODO: 记录详细日志用于回放`。

- **图形界面/控制台：** *（P1计划）* 目前机器人通过手动启动/停止，缺乏运行时的交互界面。未来可以开发简单的Web控制台或桌面GUI，实时显示机器人的状态（当前监听的策略、最近信号、持仓情况等），并允许用户手动触发操作（如平仓所有持仓、暂停/恢复监听等）。为此，P0 在架构上可考虑使用多线程或异步I/O，为将来接口预留空间。例如，主控逻辑可以运行在后台线程，主线程将来可以跑一个Flask网络服务。虽然暂不实现，但在代码中可以标记`# TODO: Web控制台接口`或者将关键状态存入某个全局状态对象，方便将来界面读取。

- **完善风控策略：** P0 的 RiskGuard 逻辑相对简洁，未来可以加入更多**风险控制和策略管理**功能。例如：

    - 根据账户资金动态调整下单手数（仓位管理优化）。

    - 针对不同策略编号设置不同的风控规则和下单参数（策略模块化），如某策略信号只交易较小仓位。

    - 引入止损止盈机制：P1 可以扩展 OrderInfo，风控模块计算合理的止损价和目标价，并通过券商API下达 OCO 单（One-Cancels-Other）或附加条件单。这需要在BrokerAdapter接口增加支持，并在风控中实现计算逻辑。

    - 多信号协调：如果同时收到多个策略对同一标的发出相反信号，如何处理？这些复杂场景可在P1+阶段处理。

- **多线程与性能优化：** P0 假定信号不频繁，单线程处理即可。若未来需要应对高频信号或多标的并行交易，可引入多线程/异步处理架构。例如，每收到一个警报启动一个线程处理交易，或使用异步io避免阻塞监听。为此可在 P0 代码中将处理逻辑和监听解耦，标记`# TODO：优化并发性能`，以便后续改造。

- **异常处理与告警：** P0 对异常简单处理（记录日志、跳过）。后续可加强异常处理策略，如在多次下单失败后通知用户（例如通过 Discord 发送一条提醒消息或邮件通知）。可以在主控模块设计一个通知函数（P0留空实现），在重要异常时调用。标记`# TODO：异常通知机制`。

- **支持更多消息来源和类型：** 将来可能不止来自Discord的警报，也可能接入其他消息源（例如Telegram频道、HTTP接口等）或者处理不同类型的信号（如新闻事件）。当前 Listener 和 Parser 设计较为通用，P1 阶段可以通过增加新 Listener 模块实现多源监听，通过在 Parser 中判断消息类型调用不同解析器来实现扩展。这部分在设计中已有一定体现，只需在代码中保持模块化，新增来源时实现对应模块即可，无需大改架构。

在代码实现中，建议使用 `TODO` 注释和文档，清晰标明以上扩展项的位置。例如，在风控判断位置添加“`# TODO：基于策略动态调整 quantity`”，在下单返回处添加“`# TODO：将执行结果写入数据库`”等。这样既能提醒开发者后续完善，又确保P0交付的代码具备良好的可读性和可扩展性，为迈向P1做好铺垫。

综上，本方案详细规划了交易机器人P0阶段的模块划分和接口设计，确保基本功能闭环实现的同时，为模拟/实盘切换、多券商支持和后续扩展预留了足够的灵活性。开发者可以据此结构开始编码，实现各模块功能并进行集成测试。待P0验证稳定后，再逐步按上述TODO要点扩充功能，完善成为一个健壮的自动交易系统。

Diagram

- Automatically receive and parse quantitative alerts from Discord, and execute trades based on these alerts.
- Provide backtesting functionality to validate strategies using historical data.

Non-functional requirements

- Performance: The system must quickly process data and execute trades with minimal latency.
- Cost: Operational expenses must be kept cost-effective and within budget.

Recommended solution

Github repo

https://github.com/Leslie-mo/MultiAgent-AutoTrading