

User:

```
class Exchange: def __init__(self, trade_symbols, leverage=20, maker_fee=0.0002,taker_fee=0.0005,log='',initial_balance=100): self.initial_balance = initial_balance #初始的资产 self.taker_fee = taker_fee self.maker_fee = maker_fee self.leverage = leverage self.trade_symbols = trade_symbols self.date = '' self.log = log self.df = pd.DataFrame() self.account = {'USDT':{'realised_profit':0, 'margin':0, 'unrealised_profit':0, 'total':initial_balance, 'leverage':0, 'fee':0,'maker_fee':0,'taker_fee':0}} for symbol in trade_symbols: self.account[symbol] = {'amount':0, 'hold_price':0, 'value':0, 'price':0, 'realised_profit':0, 'margin':0, 'unrealised_profit':0,'fee':0} def Trade(self, symbol, direction, price, amount, msg='', maker=True): if (self.date and symbol == self.log) or self.log == 'all': print('%-26s%-15s%-5s%-10.8s%-8.6s %s'%(str(self.date)[:24], symbol, 'buy' if direction == 1 else 'sell', price, amount, msg)) cover_amount = 0 if direction*self.account[symbol]['amount'] >=0 else min(abs(self.account[symbol]['amount']), amount) open_amount = amount - cover_amount if maker: self.account['USDT']['realised_profit'] -= price*amount*self.maker_fee #扣除手续费 self.account['USDT']['maker_fee'] += price*amount*self.maker_fee self.account['USDT']['fee'] += price*amount*self.maker_fee self.account[symbol]['fee'] += price*amount*self.maker_fee else: self.account['USDT']['realised_profit'] -= price*amount*self.taker_fee #扣除手续费 self.account['USDT']['taker_fee'] += price*amount*self.taker_fee self.account['USDT']['fee'] += price*amount*self.taker_fee self.account[symbol]['fee'] += price*amount*self.taker_fee if cover_amount > 0: #先平仓 self.account['USDT']['realised_profit'] += -direction*(price - self.account[symbol]['hold_price'])*cover_amount #利润 self.account['USDT']['margin'] -= cover_amount*self.account[symbol]['hold_price']/self.leverage #释放保证金 self.account[symbol]['realised_profit'] += -direction*(price - self.account[symbol]['hold_price'])*cover_amount self.account[symbol]['amount'] -= -direction*cover_amount self.account[symbol]['margin'] -= cover_amount*self.account[symbol]['hold_price']/self.leverage self.account[symbol]['hold_price'] = 0 if self.account[symbol]['amount'] == 0 else self.account[symbol]['hold_price'] if open_amount > 0: total_cost = self.account[symbol]['hold_price']*direction*self.account[symbol]['amount'] + price*open_amount total_amount = direction*self.account[symbol]['amount']+open_amount self.account['USDT']['margin'] += open_amount*price/self.leverage self.account[symbol]['hold_price'] = total_cost/total_amount self.account[symbol]['amount'] += direction*open_amount self.account[symbol]['margin'] += open_amount*price/self.leverage self.account[symbol]['unrealised_profit'] = (price - self.account[symbol]['hold_price'])*self.account[symbol]['amount'] self.account[symbol]['price'] = price self.account[symbol]['value'] = abs(self.account[symbol]['amount'])*price def Buy(self, symbol, price, amount, msg='', maker=False): self.Trade(symbol, 1, price, amount, msg, maker) def Sell(self, symbol, price, amount, msg='', maker=False): self.Trade(symbol, -1, price, amount, msg,maker) def Update(self, date, symbols, close_price): #对资产进行更新 self.date = date self.close = close_price self.account['USDT']['unrealised_profit'] = 0 for symbol in symbols: self.account[symbol]['unrealised_profit'] = (close_price[symbol] - self.account[symbol]['hold_price'])*self.account[symbol]['amount'] self.account[symbol]['price'] = close_price[symbol] self.account[symbol]['value'] = abs(self.account[symbol]['amount'])*close_price[symbol] self.account['USDT']['unrealised_profit'] += self.account[symbol]['unrealised_profit'] self.account['USDT']['total'] = round(self.account['USDT']['realised_profit'] + self.initial_balance + self.account['USDT']['unrealised_profit'],6) self.account['USDT']['leverage'] = round(self.account['USDT']['margin']*self.leverage/self.account['USDT']['total'],4), 这是我在网上看到的一个非常简单的回测类，你看能不能再改的简单点
```

ChatGPT:

下面给你一个更简洁的期货/永续合约回测账户类，保留你关心的核心要素（多空头寸、均价、已实现/未实现盈亏、maker/taker 手续费、杠杆与占用保证金），但去掉了逐品种冗余字段与复杂更新逻辑，只用少量状态即可完成结算。

```
python
from typing import Dict

class SimpleExchange:
    """
```

极简衍生品账户模型（逐合约线性计价）：

- 支持多品种多/空持仓（签名持仓：>0 多，<0 空），加权更新持仓均价
- 交易时收取 maker/taker 手续费（按名义额）
- 平仓部分实时结算已实现盈亏到现金（cash），开仓不动现金（仅计手续费）
- mark() 根据给定收盘价/最新价计算未实现盈亏、总权益、占用保证金与当前杠杆

"""

```
def __init__(
    self,
    symbols,
    leverage: float = 20.0,
    maker_fee: float = 2e-4,
    taker_fee: float = 5e-4,
    initial_balance: float = 100.0,
):
    self.symbols = list(symbols)
    self.leverage = float(leverage)
    self.maker_fee = float(maker_fee)
    self.taker_fee = float(taker_fee)

    # 账户核心状态（尽量少）:
    self.cash: float = float(initial_balance)    # 现金=初始资金-手续费+已实现PnL
    self.fees: float = 0.0                      # 累计手续费
    self.realized: float = 0.0                  # 累计已实现PnL（统计用，已计入 cash）

    # 分品种最小必要状态
    self.pos: Dict[str, float] = {s: 0.0 for s in self.symbols}    # 签名持仓张数/币数
    self.avg: Dict[str, float] = {s: 0.0 for s in self.symbols}    # 当前持仓均价（绝对价）
    self.last: Dict[str, float] = {s: 0.0 for s in self.symbols}  # 最新价（用于mark）

# --- 基本下单 ---
def trade(self, symbol: str, side: int, price: float, qty: float, maker: bool=False):
    """
    side: +1=买入（多向），-1=卖出（空向）
    qty : 成交数量（币/张）
    price: 成交价
    maker: 决定手续费率
    """
    assert symbol in self.pos, f"未知交易品种 {symbol}"
    assert side in (1, -1), "side 只能是 +1 或 -1"
    assert qty > 0 and price > 0, "qty/price 必须为正"

    fee_rate = self.maker_fee if maker else self.taker_fee
    fee = price * qty * fee_rate
```

```

self.fees += fee
self.cash -= fee # 手续费直接扣现金

p = self.pos[symbol]
a = self.avg[symbol]

# 先平仓 (若方向相反)
closing = min(abs(p), qty) if p * side < 0 else 0.0
realized = 0.0
if closing > 0:
    # 已实现PnL: (成交价-均价)*已平数量*sign(原持仓)
    realized = (price - a) * closing * (1.0 if p > 0 else -1.0)
    self.realized += realized
    self.cash += realized # 结算到现金
    # 更新持仓
    p = p + side * closing # 向0靠拢 (不会跨越0)

# 再开仓 (若有剩余数量)
opening = qty - closing
if opening > 0:
    # 若目前持仓为0, 新的均价就是本次成交价; 否则做加权均价
    if p == 0.0:
        a = price
    else:
        # 只有当新增方向与当前持仓同向才加权 (此处必同向)
        new_abs = abs(p) + opening
        a = (abs(p) * a + opening * price) / new_abs
    p = p + side * opening

# 写回
self.pos[symbol] = p
self.avg[symbol] = a
self.last[symbol] = price # 用成交价也可作为最新价的近似

def buy(self, symbol: str, price: float, qty: float, maker: bool=False):
    self.trade(symbol, +1, price, qty, maker)

def sell(self, symbol: str, price: float, qty: float, maker: bool=False):
    self.trade(symbol, -1, price, qty, maker)

# --- 账户估值 ---
def mark(self, prices: Dict[str, float]) -> Dict[str, float]:
    """
    用给定价格字典进行估值, 返回账户快照。

```

```

"""
for s, px in prices.items():
    if s in self.last and px > 0:
        self.last[s] = px

# 未实现PnL: sum( (last-avg) * pos_sign * |pos| ) = sum( (last-avg) * pos )
unreal = 0.0
notional = 0.0
for s in self.symbols:
    if self.pos[s] != 0 and self.avg[s] > 0 and self.last[s] > 0:
        unreal += (self.last[s] - self.avg[s]) * self.pos[s]
    if self.last[s] > 0:
        notional += abs(self.pos[s]) * self.last[s]

equity = self.cash + unreal
margin_req = notional / self.leverage if self.leverage > 0 else 0.0
used_leverage = (notional / equity) if equity > 0 else float("inf")

return {
    "cash": round(self.cash, 8),
    "fees": round(self.fees, 8),
    "realized_pnl": round(self.realized, 8),
    "unrealized_pnl": round(unreal, 8),
    "equity": round(equity, 8),
    "notional": round(notional, 8),
    "margin_required": round(margin_req, 8),
    "used_leverage": round(used_leverage, 6),
    "positions": {s: {"pos": self.pos[s], "avg": self.avg[s], "last": self.last[s]} for s in self.symbols},
}

```

简化点说明（和你原版相比）

- 账户层面只保留 **cash / fees / realized** 三个全局量，分品种只保留 **pos/avg/last** 三个字段；平开仓合并 `trade()` 一步完成。
- **保证金** 不再逐笔累加/释放，而是估值时按当前名义额 `notional/leverage` 直接计算（更直观，也更少状态）。
- **已实现盈亏** 在平仓瞬间结转进 `cash`，未实现盈亏由 `mark()` 统一计算；打印日志等功能去掉，外部需要的话可在 `trade()/mark()` 之外自行加。

User:

用你写的回测类生成一个统计套利策略模版

ChatGPT:

下面给你一个可直接套用的统计套利（配对交易/价差交易）策略模板，已集成我上条消息给的 SimpleExchange 回测账户类。
模板特点：

- 支持两资产配对 (y/x)，滚动估计对冲比率 β 与截距 α ；
- 基于价差的z-score 入场/出场；
- 目标头寸→下单的完整执行环；
- 风险控制（最大杠杆、止损z阈、资金分配比例）；
- 绩效统计（权益曲线、回撤、夏普等）。

把下面整段代码粘进一个 .py 文件或 Notebook 即可运行（自带一个合成数据示例）。你也可以把数据换成真实行情，只要按注释准备好即可。

```
python
# ===== 依赖 =====
import numpy as np
import pandas as pd
from dataclasses import dataclass
from typing import Dict, Tuple

# ===== 回测账户：SimpleExchange（来自上条消息，原样贴出以便自包含） =====
class SimpleExchange:
    def __init__(self, symbols, leverage=20.0, maker_fee=2e-4, taker_fee=5e-4, initial_balance=100.0):
        self.symbols = list(symbols)
        self.leverage = float(leverage)
        self.maker_fee = float(maker_fee)
        self.taker_fee = float(taker_fee)

        self.cash = float(initial_balance)
        self.fees = 0.0
        self.realized = 0.0

        self.pos: Dict[str, float] = {s: 0.0 for s in self.symbols}
        self.avg: Dict[str, float] = {s: 0.0 for s in self.symbols}
        self.last: Dict[str, float] = {s: 0.0 for s in self.symbols}

    def trade(self, symbol: str, side: int, price: float, qty: float, maker: bool = False):
        assert symbol in self.pos
        assert side in (1, -1)
        assert qty > 0 and price > 0
```

```
fee_rate = self.maker_fee if maker else self.taker_fee
fee = price * qty * fee_rate
self.fees += fee
self.cash -= fee

p = self.pos[symbol]
a = self.avg[symbol]

closing = min(abs(p), qty) if p * side < 0 else 0.0
if closing > 0:
    realized = (price - a) * closing * (1.0 if p > 0 else -1.0)
    self.realized += realized
    self.cash += realized
    p = p + side * closing

opening = qty - closing
if opening > 0:
    if p == 0.0:
        a = price
    else:
        new_abs = abs(p) + opening
        a = (abs(p) * a + opening * price) / new_abs
    p = p + side * opening

self.pos[symbol] = p
self.avg[symbol] = a
self.last[symbol] = price

def buy(self, symbol: str, price: float, qty: float, maker: bool=False):
    self.trade(symbol, +1, price, qty, maker)

def sell(self, symbol: str, price: float, qty: float, maker: bool=False):
    self.trade(symbol, -1, price, qty, maker)

def mark(self, prices: Dict[str, float]) -> Dict[str, float]:
    for s, px in prices.items():
        if s in self.last and px > 0:
            self.last[s] = px

unreal = 0.0
notional = 0.0
for s in self.symbols:
    if self.pos[s] != 0 and self.avg[s] > 0 and self.last[s] > 0:
        unreal += (self.last[s] - self.avg[s]) * self.pos[s]
    if self.last[s] > 0:
        notional += abs(self.pos[s]) * self.last[s]
```

```

equity = self.cash + unreal
margin_req = notional / self.leverage if self.leverage > 0 else 0.0
used_leverage = (notional / equity) if equity > 0 else float("inf")

return {
    "cash": round(self.cash, 8),
    "fees": round(self.fees, 8),
    "realized_pnl": round(self.realized, 8),
    "unrealized_pnl": round(unreal, 8),
    "equity": round(equity, 8),
    "notional": round(notional, 8),
    "margin_required": round(margin_req, 8),
    "used_leverage": round(used_leverage, 6),
    "positions": {s: {"pos": self.pos[s], "avg": self.avg[s], "last": self.last[s]} for s in self.symbols},
}

# ===== 策略参数 =====
@dataclass
class PairStatArbParams:
    lookback_beta: int = 300          # 滚动窗口估计对冲比率  $\beta$ /截距  $\alpha$  的长度
    lookback_z: int = 300             # 滚动窗口计算价差均值/标准差
    entry_z: float = 1.5              # 入场阈值 ( $|z| > \text{entry\_z}$ )
    exit_z: float = 0.5               # 出场阈值 ( $|z| < \text{exit\_z}$ )
    stop_z: float = 3.5               # 止损阈值 ( $|z| > \text{stop\_z}$  则强制平仓)
    capital_frac: float = 0.2         # 每个bar用于该配对的资金比例 (相对权益)
    max_leverage: float = 5.0         # 账户使用杠杆上限 (超过则按比例缩小目标持仓)
    use_maker: bool = False           # 下单类型 (模板默认用 taker 更贴近现实)
    min_trade_qty: float = 1e-6       # 最小下单数量 (防抖)
    include_intercept: bool = True    #  $\beta$ 估计是否包含截距 (推荐 True)

# ===== 工具函数: 滚动OLS估计  $\alpha, \beta$  ( $y \sim \alpha + \beta x$ ) =====
def rolling_ols_alpha_beta(y: pd.Series, x: pd.Series, win: int, include_intercept: bool=True) -> Tuple[pd.Series, pd.Series]:
    """
    返回: alpha, beta 的滚动估计 (与 y/x 对齐, 前 win-1 为 NaN)
    采用闭式解: beta = Cov/Var, alpha = mean_y - beta*mean_x
    """
    mx = x.rolling(win).mean()
    my = y.rolling(win).mean()
    vx = (x.rolling(win).var(ddof=0)).replace(0, np.nan)
    cov = ( (x* y).rolling(win).mean() - mx*my )

    beta = cov / vx
    if not include_intercept:

```

```

        alpha = pd.Series(0.0, index=y.index)
    else:
        alpha = my - beta * mx
    return alpha, beta

# ===== 策略本体 (两资产配对) =====
class PairStatArbStrategy:
    """
    y = symbol_y, x = symbol_x
    价差: spread = y - (alpha + beta * x)
    信号: zscore(spread)
        z > entry_z -> 做空价差 (short y, long x)
        z < -entry_z -> 做多价差 (long y, short x)
        |z| < exit_z -> 平仓
    """
    def __init__(self, exch: SimpleExchange, symbol_y: str, symbol_x: str, params: PairStatArbParams):
        self.exch = exch
        self.y = symbol_y
        self.x = symbol_x
        self.p = params

        self.alpha = None
        self.beta = None
        self.spread = None
        self.z = None

        # 内部状态
        self._equity_series = [] # 记录每步权益
        self._snapshots = []     # 记录账户快照
        self._signal = 0         # -1 空价差, +1 多价差, 0 空仓

    def _compute_indicators(self, py: pd.Series, px: pd.Series):
        a, b = rolling_ols_alpha_beta(py, px, self.p.lookback_beta, self.p.include_intercept)
        spread = py - (a + b * px)

        mu = spread.rolling(self.p.lookback_z).mean()
        sd = spread.rolling(self.p.lookback_z).std(ddof=0)
        z = (spread - mu) / sd

        self.alpha, self.beta, self.spread, self.z = a, b, spread, z

    def _target_pair_position(self, price_y: float, price_x: float, beta: float, z_now: float, equity: float) -> Tuple[float, float]:
        """
        给定价格、 $\beta$ 、 $z$ 值与当前权益, 计算目标持仓 (签名数量), 实现名义额近似配对中性:
        
$$\text{Notional}_y \approx \text{Notional}_x * |\beta|$$


```



```
"""
# 无信号/出场: 目标0
if np.isnan(z_now) or np.isnan(beta):
    return 0.0, 0.0

# 入场/反转/出场信号
sig = 0
if abs(z_now) > self.p.stop_z:
    sig = 0 # 超阈直接强平
elif z_now > self.p.entry_z:
    sig = -1 # 做空价差: short y, long x
elif z_now < -self.p.entry_z:
    sig = +1 # 做多价差: long y, short x
elif abs(z_now) < self.p.exit_z:
    sig = 0 # 价差回归, 退出

# 目标名义额 (两腿合计)
alloc = max(self.p.capital_frac, 0.0) * max(equity, 0.0)
if alloc <= 0 or sig == 0:
    return 0.0, 0.0

beta_abs = abs(beta) if np.isfinite(beta) else 1.0
# 按 y : (|beta| * x) 的名义额配比
denom = (price_y + beta_abs * price_x)
if denom <= 0:
    return 0.0, 0.0

qty_pair = alloc / denom # 一“组”头寸的数量规模
# 多价差: +y , -beta*x ; 空价差: -y , +beta*x
qty_y = sig * qty_pair
qty_x = -sig * beta * qty_pair # 注意 beta 的符号 (对冲比率可能为负)

return qty_y, qty_x

def _rebalance_to_target(self, price_y: float, price_x: float, tgt_y: float, tgt_x: float):
    """根据目标头寸与当前头寸差额下单"""
    cur_y = self.exch.pos[self.y]
    cur_x = self.exch.pos[self.x]

    dy = tgt_y - cur_y
    dx = tgt_x - cur_x

    # 最小成交量过滤
    if abs(dy) >= self.p.min_trade_qty:
```

```
        if dy > 0:
            self.exch.buy(self.y, price_y, abs(dy), maker=self.p.use_maker)
        else:
            self.exch.sell(self.y, price_y, abs(dy), maker=self.p.use_maker)

    if abs(dx) >= self.p.min_trade_qty:
        if dx > 0:
            self.exch.buy(self.x, price_x, abs(dx), maker=self.p.use_maker)
        else:
            self.exch.sell(self.x, price_x, abs(dx), maker=self.p.use_maker)

    def _enforce_leverage_cap(self, price_y: float, price_x: float):
        """若超过最大杠杆上限, 则按比例减仓"""
        snap = self.exch.mark({self.y: price_y, self.x: price_x})
        if not np.isfinite(snap["used_leverage"]) or snap["used_leverage"] >= self.p.max_leverage:
            return

        scale = self.p.max_leverage / max(snap["used_leverage"], 1e-12)
        # 目标等比例缩小现有仓位
        tgt_y = self.exch.pos[self.y] * scale
        tgt_x = self.exch.pos[self.x] * scale
        self._rebalance_to_target(price_y, price_x, tgt_y, tgt_x)

    def step(self, t, price_y: float, price_x: float, alpha: float, beta: float, z_now: float):
        # 根据信号做目标持仓
        snap_before = self.exch.mark({self.y: price_y, self.x: price_x})
        tgt_y, tgt_x = self._target_pair_position(price_y, price_x, beta, z_now, snap_before["equity"])
        self._rebalance_to_target(price_y, price_x, tgt_y, tgt_x)
        # 杠杆上限校验
        self._enforce_leverage_cap(price_y, price_x)
        # 记账
        snap = self.exch.mark({self.y: price_y, self.x: price_x})
        self._equity_series.append((t, snap["equity"]))
        self._snapshots.append(snap)

    def run(self, df: pd.DataFrame):
        """
        df: 行为时间索引, 列为 {self.y, self.x} 的价格 (收盘/中价)
        """
        assert self.y in df.columns and self.x in df.columns
        py = df[self.y].astype(float)
        px = df[self.x].astype(float)

        self._compute_indicators(py, px)
```

```

    for t, price_y, price_x, a, b, z_now in zip(
        df.index, py.values, px.values,
        self.alpha.values, self.beta.values, self.z.values
    ):
        self.step(t, price_y, price_x, a, b, z_now)

    eq = pd.Series({t: v for t, v in self._equity_series})
    return eq, pd.DataFrame(self._snapshots, index=df.index)

# ===== 绩效评估 =====
def perf_stats(equity: pd.Series, bar_seconds: int = 60):
    equity = equity.dropna()
    ret = equity.pct_change().dropna()
    if len(ret) == 0:
        return {"sharpe": np.nan, "max_dd": np.nan, "return": np.nan, "vol": np.nan}

    per_year = int(round(365 * 24 * 3600 / bar_seconds))
    mu = ret.mean() * per_year
    sigma = ret.std(ddof=0) * np.sqrt(per_year)
    sharpe = mu / (sigma + 1e-12)

    cum = (1 + ret).cumprod()
    peak = cum.cummax()
    dd = (cum / peak - 1.0).min()

    total_return = equity.iloc[-1] / equity.iloc[0] - 1.0
    return {
        "sharpe": float(sharpe),
        "max_dd": float(dd),
        "return": float(total_return),
        "vol": float(sigma),
    }

# ===== 示例：合成两资产并运行回测（替换为你的实盘数据即可） =====
if __name__ == "__main__":
    np.random.seed(42)

    # 1) 生成合成数据: x 随机游走,  $y \approx \alpha + \beta x$  + 小噪声 (弱协整)
    n = 10_000
    idx = pd.date_range("2025-01-01", periods=n, freq="T") # 1分钟bar
    x = np.cumsum(np.random.normal(0, 0.05, size=n)) + 100
    beta_true = 1.2
    alpha_true = 0.8
    noise = np.random.normal(0, 0.2, size=n)
    y = alpha_true + beta_true * x + noise

```

```
# 2) 组装价格表（真实使用中：把这里替换为真实的分钟或秒级收盘价/中间价）
df = pd.DataFrame({"Y": y, "X": x}, index=idx)

# 3) 初始化账户与策略
symbols = ["Y", "X"]
exch = SimpleExchange(symbols, leverage=10.0, taker_fee=5e-4, maker_fee=2e-4, initial_balance=1_000.0)
params = PairStatArbParams(
    lookback_beta=300,
    lookback_z=300,
    entry_z=1.5,
    exit_z=0.5,
    stop_z=3.5,
    capital_frac=0.2,
    max_leverage=5.0,
    use_maker=False,
    min_trade_qty=1e-6,
    include_intercept=True,
)
strat = PairStatArbStrategy(exch, symbol_y="Y", symbol_x="X", params=params)

# 4) 运行回测
equity, snaps = strat.run(df)

# 5) 输出绩效
stats = perf_stats(equity, bar_seconds=60)
print("Performance:", stats)
print("Final snapshot:", snaps.iloc[-1].to_dict())

# 若需要可视化（可选）：
try:
    import matplotlib.pyplot as plt
    plt.figure(figsize=(8, 3))
    equity.plot()
    plt.title("Equity Curve")
    plt.tight_layout()
    plt.show()
except Exception as e:
    pass
```

如何用在真实数据上

- 准备一个 DataFrame：df，索引为时间（升序），列名必须是你的两个交易对符号，例：{"BTCUSDT_PERP", "ETHUSDT_PERP"}，列值为该bar的收盘价或中间价。
- 把 symbols = ["Y", "X"] 换成你的两个真实符号，并在 df 里用相同列名。

- 调整 PairStatArbParams:
 - lookback_beta / lookback_z: 根据你的频率（秒/分）和稳定性调整。
 - entry_z/exit_z/stop_z: 可从 1.5/0.5/3.5 起步，再做灵敏度测试。
 - capital_frac: 单bar 可用资金占比，配合 max_leverage 控制风险。
 - use_maker: 如希望仅模拟挂单成交，设 True（但要明白真实挂单成交概率问题）。

拓展位（可按需加）

- 多配对/多资产：把 PairStatArbStrategy 实例化多份（不同 y/x），或写个“调度器”按轮询顺序逐策略 step。
- 动态 β 估计方式：改为滚动回归 with 半衰期权重、卡尔曼滤波、或Engle-Granger 协整检验 + 残差AR(1)。
- 成交模型：加入滑点模型、挂单成交概率/队列模型、部分成交与撤单重试。
- 约束：持仓上限（张/名义额）、单笔最大下单量、跨交易所费率差异等。

需要我把它改造成多配对并行框架或换成卡尔曼滤波动态 β 版本吗？你丢一个真实的两品种价序到我这，我直接替你跑一版对比参数也可以。

User:

lookback_beta / lookback_z: entry_z/exit_z/stop_z: capital_frac, 这几个参数是什么意思，详细讲讲

ChatGPT:

下面把这 3 组参数讲清楚：它们分别控制模型估计的“记忆长度”、进出场的判别强度和每次投入的资金规模，共同决定了频率、胜率、持仓时间与风控强度。

1) lookback_beta 与 lookback_z

lookback_beta（估计对冲比率 β 与截距 α 的窗口）

- 做的事：在长度为 lookback_beta 的滚动窗口里，对 y_t 与 x_t 做闭式 OLS ($y_t \approx \alpha + \beta x_t$)，得到当前时刻的 α_t, β_t 。
- 越短 $\rightarrow \beta/\alpha$ 更快跟随结构变化，但更噪（对瞬时价差抖动更敏感，易误判）。
- 越长 $\rightarrow \beta/\alpha$ 更稳、配对名义额更匹配，但反应慢（错过结构变更，回归更迟钝）。

- 经验区间（1 分钟 bar）：
 - 高频/快变：100–300
 - 正常/日内：300–600
 - 稳定/多日：600–1500
- 小技巧：如果价差结构在不同阶段有轻微漂移，lookback_beta 可略短于 lookback_z（让 β 更灵活；z 的均值/方差更稳）。

lookback_z（计算价差 z-score 的窗口）

- 做的事：先算残差 $s_t = y_t - (\alpha_t + \beta_t x_t)$ $st = yt - (\alpha t + \beta txt)$ 。在 lookback_z 窗里计算 s_tst 的滚动均值与标准差，然后

$$z_t = \frac{s_t - \mu_t}{\sigma_t}.$$

$$zt = \sigma tst - \mu t.$$

- 越短 → 触发更频繁（噪声大、标准差估小），胜率可能下降，交易/费率更多。
- 越长 → 信号更稀疏（波动被平滑），但每次 z 突破更有“显著性”。
- 经验：让 lookback_z \geq lookback_beta，常见取法是 相等 或 略大 1–2 倍。
例：lookback_beta=300，lookback_z=300~600。
- 变体：可用 EWMA/半衰期 代替等权滚动，让 z 对异常点更鲁棒。

2) entry_z / exit_z / stop_z

这 3 个阈值决定“何时开、何时平、何时止损”。它们都在 价差标准化后的尺度 上设定，具有跨品种可比性。

entry_z（入场阈值）

- 逻辑：当 $|z_t| > entry_z|zt| > entry_z$ 认为价差偏离均衡较大，做均值回归：
 - $z_t > entry_zzt > entry_z$ ：做空价差（卖 yy、买 xx）
 - $z_t < -entry_zzt < -entry_z$ ：做多价差（买 yy、卖 xx）
- 越小 → 更频繁入场，持仓更密集，但噪声/费用压力更大。
- 越大 → 机会更稀疏，但“每次偏离更极端”，胜率/单笔回归空间更高。

- 粗略频率感（假设 z 近似标准正态）：
 - $|z| > 1.0$ $|z| > 1.0$ 的概率 $\approx 31.7\%$
 - $|z| > 1.5$ $|z| > 1.5$ 的概率 $\approx 13.4\%$
 - $|z| > 2.0$ $|z| > 2.0$ 的概率 $\approx 4.6\%$
 - $|z| > 2.5$ $|z| > 2.5$ 的概率 $\approx 1.2\%$
 - $|z| > 3.0$ $|z| > 3.0$ 的概率 $\approx 0.27\%$
（数值用于相对比较触发频率，实际分布可能肥尾）

exit_z（出场阈值）

- 逻辑：当 $|z_t| < \text{exit_z}|z_t| < \text{exit_z}$ 认为价差已“回到均衡带”，主动平仓锁定回归收益。
- 常设为 小于 entry_z，形成进宽出窄的“回归区间”。
- 越小 \rightarrow 更快落袋为安，但容易“过早离场”丢失尾部回归收益。
- 经验搭配：
 - entry_z=1.0~1.5, exit_z=0.25~0.6
 - entry_z=2.0, exit_z=0.5~0.8

stop_z（止损阈值）

- 逻辑：当 $|z_t| > \text{stop_z}|z_t| > \text{stop_z}$ 说明可能发生结构断裂/协整失效，立刻清仓。
- 应显著大于 entry_z，常见 3.0–5.0。
- 也可用分位数法：例如历史 $|z||z|$ 的 99.5% 分位作为 stop_z，可随时间自适应。

阈值与费用/滑点的关系

- 一次配对交易是双腿同时成交，用 taker 时单边 5bp、双边约 10bp（仅手续费，未含滑点）。
- 因此 entry_z 不能太小：期望回归幅度（以 z 的单位 $\times \sigma$ ）应覆盖双边费用+滑点，再留一定 Alpha 缓冲。
- 在更高频/更高费率品上，适当提高 entry_z 或 拉长 lookback_z（让 σ 更稳）有帮助。

3) capital_frac（每步用于该配对的名义额占当前权益的比例）

- 做的事：我们把当前账户权益 equity 的一部分

$$\text{alloc} = \text{capital_frac} \times \text{equity}$$

$$\text{alloc} = \text{capital_frac} \times \text{equity}$$

分给该配对，然后按“y:|β|xy : |β|x”的名义额比例拆成两腿下单。

- 重要结论：在这个模板里，下单后的名义额 $\approx \approx \text{alloc}$ ，因此

$$\text{used_leverage} = \frac{\text{notional}}{\text{equity}} \approx \text{capital_frac}.$$

$$\text{used_leverage} = \text{equitynotional} \approx \text{capital_frac}.$$

也就是说，**capital_frac** 就是你愿意在该配对上使用的“账户杠杆占比”（独立于交易所合约杠杆设置）。

举例：equity=10,000, capital_frac=0.3 → 该配对本步名义额约 3,000, used_leverage≈0.3。

- 与 SimpleExchange.leverage（账户里设的 10x、20x）关系：
 - SimpleExchange.leverage 仅用于保证金需求报告：margin_required = notional / leverage。
 - 我们的风控用 max_leverage（策略参数）去强制缩放仓位，防止过度放大。
- 选取建议：
 - 单配对、日内：0.1–0.4 常见；想更保守就更小。
 - 多配对并行：确保所有配对名义额之和 / 权益 ≤ 你接受的账户总使用杠杆（比如 1.0–2.0）。
 - 若波动升高或滑点上升，可动态下调 capital_frac。

快速调参范式（给 1 分钟线的起点）

- 第一步：lookback_beta=300, lookback_z=300~600。
看看 z 的分布是否近似 0 均值、稳定（做个直方图、滚动均值/方差图）。
- 第二步：entry_z=1.5, exit_z=0.5, stop_z=3.5。
若信号太少 → 降低 entry_z；若胜率/净利差 → 提高 entry_z 或增大 lookback_z。
- 第三步：capital_frac=0.2 起步，观察 used_leverage 是否被 max_leverage 触发缩仓；若没有且回测回撤小，可缓慢上调。

- 第四步（费率/滑点体感）：若净收益被费用吃掉，优先 提高 **entry_z** 或 加大 **lookback_z**，其次再考虑 maker（但需建成交模型）。

常见失败与修复

- β 漂移导致对冲失配：降低 `lookback_beta` 或改用 半衰期回归/卡尔曼滤波。
- `z` 过于敏感，频繁进出：提高 `lookback_z` 或提高 `entry_z`、降低 `exit_z` 差距。
- 极端行情“越止损越涨”：提高 `stop_z` 或改为分位数止损（随波动自适应），并加“冷却时间/重启条件”。
- 名义额过大触发缩仓：降低 `capital_frac`，或提高策略层 `max_leverage`（前提是你接受的整体杠杆更高）。

如果你告诉我数据频率、标的对（比如 BTC/ETH 永续）、以及你目标的交易频率/杠杆上限，我可以按这个范式直接给一组更合适的初始参数，并加上 半衰期回归 或 卡尔曼 β 的版本，让 `lookback_beta/lookback_z` 不必是固定窗口。

User:

对 y_t 与 x_t 做闭式 OLS，这里的 y_t 和 x_t 是什么

ChatGPT:

好问题！这里的 y_t 和 x_t 是在做配对交易（统计套利）时，我们选取的**两只资产的价格序列**。

1. 背景：闭式 OLS 在这里的作用

我们在配对交易里假设两只资产之间存在协整关系，可以近似写作：

$$y_t \approx \alpha + \beta x_t + \varepsilon_t,$$

$$y_t \approx \alpha + \beta x_t + \varepsilon_t,$$

其中：

- y_t ：被解释变量（dependent variable）
- x_t ：解释变量（independent variable）

- α : 截距
- β : 对冲比率 (hedge ratio)
- ε_t : 残差 (spread, 价差)

OLS (普通最小二乘) 就是在滚动窗口里不断拟合 α, β , 保证残差 ε_t 尽量稳定。

2. 在策略里的具体含义

- y_t : 配对里的一个品种, 比如 ETHUSDT 的价格 (我们设为“主”资产)。
- x_t : 另一个品种, 比如 BTCUSDT 的价格 (我们设为“对冲”资产)。
- 滚动 OLS: 在过去 lookback_beta 根 K 线的数据里, 用 ETH ~ BTC 的线性回归算出 β 和 α 。
- 残差价差:

$$s_t = y_t - (\alpha + \beta x_t)$$

这就是我们关心的“价差”。它应该是均值回归的。

3. 为什么要这样设

- 如果直接看 ETH-BTC 价差, BTC 涨跌 100 美元时, ETH 可能对应涨跌 5 美元, 这样“裸价差”不稳定。
- 通过 OLS 得到的 β , 其实是 ETH 对 BTC 的“动态杠杆系数”, 让残差更平稳、更接近平稳序列。
- 在交易层面:
 - 做多价差 = 买 ETH, 卖 β 张 BTC;
 - 做空价差 = 卖 ETH, 买 β 张 BTC。

这样才能保证整体是对冲后中性的套利, 而不是暴露在单边市场风险里。

4. 举个例子

假设过去 300 根 1 分钟数据里，我们回归得到：

$$y_t = \alpha + 0.07x_t + \varepsilon_t$$

$$y_t = \alpha + 0.07x_t + \varepsilon_t$$

其中：

- y_t = ETHUSDT 价格
- x_t = BTCUSDT 价格
- $\beta = 0.07$

那么交易时：

- 如果残差 s_t 偏高（ETH 相对贵了），策略会 空 ETH，多 0.07 BTC。
- 如果残差 s_t 偏低（ETH 相对便宜），策略会 多 ETH，空 0.07 BTC。

✅ 总结：在你的代码和策略里，

- y_t 和 x_t 就是选定的两只交易标的的价格序列（可以是收盘价、中间价或 mid-price）。
- 我们用它们做滚动 OLS，得到 α 和 β ，从而定义“价差”这个均值回归的交易信号。

要不要我给你画一张小图（用 Python 画价差残差和 z-score 曲线），这样你会直观地看到 $y_t, x_t \rightarrow$ OLS 拟合 \rightarrow 价差 \rightarrow z-score 的整个链条？

User:

你的模板是向量化回测吗，如果是的话，我可能需要根据历史数据自己计算比较复杂信号，那么向量化回测可能无法满足，或者我自己把信号计算好，一起融入模板，这样用向量化应该也行，你觉得哪个好一点？

ChatGPT:

短答：**不是纯向量化**。我的模板是“**指标计算向量化 + 执行逐步仿真**”的混合式： $\alpha/\beta/z$ 用滚动窗口一次性算好（向量化），但下单结算必须一根一根时间推进（路径依赖），所以用事件驱动循环。

你要做**更复杂/重历史**的信号有两条路都行，取决于是否依赖“模拟中的账户状态”。

选哪种更好？

A) 先把信号都算好再喂给执行层（推荐：纯历史函数型信号）

适用：信号只依赖过去行情/特征（可向量化/Polars/Numba 算），不依赖“是否已开仓、浮盈、回撤、是否触发过止损”等模拟状态。

优点：快、易复现实验、便于做网格搜索。

做法：把 $\alpha/\beta/z$ 或者干脆直接把目标持仓 $\text{tgt_y}/\text{tgt_x}$ 预先算成列，执行层只负责“跟仓位”。

最小改动示例（将预计算结果直接接入模板）：

```
python
# 假设你已得到以下 Series (与 df.index 对齐)
alpha_s = ...    # 也可以不提供
beta_s   = ...
z_s      = ...

# 运行时传入即可：优先用你提供的列
equity, snaps = strat.run(df.assign(__alpha=alpha_s, __beta=beta_s, __z=z_s))
```

把 `PairStatArbStrategy.run()` 改成发现 `__alpha/__beta/__z` 存在就用它们，否则才内部滚动 OLS。

更进一步：如果你直接给出 $\text{tgt_y} / \text{tgt_x}$ （每根bar目标持仓），执行层只需：

```
python
for t in df.index:
    price_y, price_x = df.loc[t, y], df.loc[t, x]
    tgt_y, tgt_x = df.loc[t, "tgt_y"], df.loc[t, "tgt_x"]
    strat._rebalance_to_target(price_y, price_x, tgt_y, tgt_x)
    strat._enforce_leverage_cap(price_y, price_x)
    snap = exch.mark({y: price_y, x: price_x})
```

这样执行层完全“哑化”，方便你把复杂度放在特征工程与信号工程里。

B) 在执行循环里现算信号（路径依赖/状态依赖信号）

适用：信号需要用到模拟状态（例如“进场后的高低点轨迹、累计手续费/浮盈、持仓持有期、分级仓位管理、触发型状态机、冷却时间”等）。

优点：能表达真实交易逻辑；能做复杂风控/再平衡；适合做市、队列成交、止损后冷却等。

做法：给 `run()` 传一个回调 `compute(t, hist, exch, state)`，每步用它产出信号/目标仓位。

接口示例（只需很小改动）：

```
python
def compute(t, hist_df, exch, state):
    # hist_df: 到 t 的历史 (含你需要的任意特征列)
    # exch: 可读到目前仓位/均价/权益/用杠杆等
    # state: 你自定义的跨步状态字典 (可记录入场价、高低点、冷却计数等)
    # 返回: alpha, beta, z, 以及可选的 target 或附加信息
    ...
    return {"alpha": a, "beta": b, "z": z, "tgt_y": tgt_y, "tgt_x": tgt_x}

def run(df, compute=None):
    state = {}
    for t in df.index:
        price_y, price_x = df.at[t, y], df.at[t, x]
        if compute:
            out = compute(t, df.loc[:t], exch, state)
            if "tgt_y" in out:
                strat._rebalance_to_target(price_y, price_x, out["tgt_y"], out["tgt_x"])
            else:
                # 否则按 (alpha,beta,z) 走默认逻辑
                ...
        strat._enforce_leverage_cap(price_y, price_x)
        snap = exch.mark({y: price_y, x: price_x})
```

这就能支持任何非线性状态机、交易后依赖、分阶段加减仓、动态费用/滑点等。

该怎么选?

- 可量化 (纯历史) → **A 路线** (预先算好列): 计算快、对比实验方便; 执行层只跟仓位。
- 强路径依赖/要读账户状态 → **B 路线** (回调现算): 表达力强、贴近实盘; 计算慢但真实。

很多人采用混合:

- 90% 可量化的特征/过滤器先离线算好 (Polars/Numba),
- 少量路径依赖 (如“进场后才启用的拖尾止损/动态权重”) 在回测循环里完成。

如果你告诉我你要做的“复杂信号”主要是哪类 (纯历史还是状态依赖), 我可以把模板朝 A 或 B 的方向**直接改造成**你可用的骨架 (包含接口与最小代码改动)。

User:

你这个“纯历史”和“状态依赖”是什么意思？

ChatGPT:

好，我把这两个词拆开讲，你就能区分了：

1. “纯历史型（history-only）信号”

指的是：

👉 信号只依赖于历史行情数据（价格、成交量、技术指标等），不依赖账户当前状态。

特征

- 输入：K线、盘口、因子、外部指标（都能预先算好）。
- 输出：直接得到某时刻的信号值（买/卖/平仓、目标仓位比例）。
- 无路径依赖：在回测里，不需要知道前一笔交易是否成功、不需要账户的盈亏情况，信号就能算出来。
- 可向量化：因为完全基于历史数据，可以用 Pandas/Polars/Numba 一次性算出全列，不必逐根 K 线循环。

例子

- 均线交叉（短均线 > 长均线 → 买入）。
- z-score 超过阈值 → 入场。
- 波动率低于某水平 → 加仓。

这些只需要历史价格序列，不需要账户“状态”（持仓、浮盈、交易后高点等）。

2. “状态依赖型（state-dependent）信号”

指的是：

👉 信号需要参考账户/交易过程的状态，即“路径依赖”。

特征

- **输入**：除了行情，还要用到账户状态：当前仓位、浮盈亏、是否触发过止损、历史持仓高低点、已实现手续费等。
- **输出**：信号可能会因为账户的不同状态而不同。
- **路径依赖**：因为交易行为会改变账户状态 → 状态又会影响未来信号 → 必须逐步仿真。
- **无法纯向量化**：必须在回测循环中，每一根 bar 更新一次状态、重新计算信号。

例子

- **动态止盈/止损**：入场后，如果浮盈超过 2%，就设置一个拖尾止损。这个拖尾止损必须知道“进场点、最高浮盈点”，所以依赖状态。
- **冷却期**：止损出局后，要等待 30 根 K 线才能再次开仓，这就依赖账户“最近是否触发过止损”。
- **仓位递增**：若持仓盈利，就加仓一半；若持仓亏损，减仓。加减仓规则直接依赖账户状态。
- **资金分配**：账户回撤超过 10% 后，降低所有配对的 `capital_frac`。

3. 两者结合

现实里，大部分基础因子是“纯历史”，比如均值回归信号、趋势突破信号；但真正落地交易，一定会加“状态依赖”：止损、止盈、资金管理、冷却期等。

所以在回测里，通常是：

- **纯历史部分**：向量化算出指标、信号列（快）。
- **状态依赖部分**：在逐步循环里调用账户状态再调整（真实）。

✅ 总结一句：

- **纯历史型** = 跟账户无关，能预先整列算出来；
- **状态依赖型** = 必须知道账户当前状态，得逐步仿真。

要不要我帮你在我之前写的那个回测模板里，加一个**状态依赖示例**（比如“进场后加拖尾止损”），这样你能看到两者结合的实际代码差别？

User:

可以的，加一个状态依赖示例（比如“进场后加拖尾止损”）

ChatGPT:

好嘞！下面是在你现有模板上**加“进场后拖尾止盈（trailing stop）”**的最小改动版。思路是：

- 仍用 z-score 做入场/出场；
- **进场后**开始跟踪“对自己有利的最佳 z 值”（best_z）；
- 一旦当前 z 从 best_z 回撤超过 **trail_z**，就触发平仓；
- 可设置 trail_min_hold（最小持有 bars）避免刚进场就被震出去。

我只贴出改动处 + 一个跑通示例，便于你直接替换。

1) 增加参数

```
python
from dataclasses import dataclass

@dataclass
class PairStatArbParams:
    lookback_beta: int = 300
    lookback_z: int = 300
    entry_z: float = 1.5
    exit_z: float = 0.5
    stop_z: float = 3.5
    capital_frac: float = 0.2
    max_leverage: float = 5.0
    use_maker: bool = False
    min_trade_qty: float = 1e-6
    include_intercept: bool = True

    # === 新增：拖尾止盈 ===
    use_trailing: bool = True          # 打开/关闭拖尾
    trail_z: float = 0.8              # 允许的z回撤幅度（单位：z）
    trail_min_hold: int = 10          # 进场后至少持有多少bar才评估拖尾
```

2) 给策略类加“状态”与拖尾逻辑

把下面这些方法/字段加到你的 PairStatArbStrategy 里 (其余不动):

```
python
class PairStatArbStrategy:
    def __init__(self, exch, symbol_y, symbol_x, params):
        ...
        self._equity_series = []
        self._snapshots = []
        self._signal = 0
        # === 新增: 记录拖尾需要的状态 ===
        self._state = {
            "in_pos": False,    # 是否持有配对头寸
            "dir": 0,           # +1=多价差, -1=空价差
            "entry_bar": None,  # 进场的 bar 序号 (或时间)
            "best_z": None,     # 进场后“对自己有利”的最佳 z
        }

    def _infer_dir_from_position(self) -> int:
        """根据当前两腿持仓推断价差方向: +1 多价差 (+y, -x·β), -1 空价差 (-y, +x·β), 0 空仓。"""
        py = self.exch.pos[self.y]
        px = self.exch.pos[self.x]
        if abs(py) < 1e-12 and abs(px) < 1e-12:
            return 0
        # 简化判断: y腿的符号代表方向 (模板下单保持这点成立)
        return 1 if py > 0 else -1

    def _on_position_change(self, now_bar_idx: int, z_now: float):
        """检测进场/清仓的状态切换, 并初始化/清理拖尾信息。"""
        dir_now = self._infer_dir_from_position()
        was_in = self._state["in_pos"]

        if not was_in and dir_now != 0:
            # 新进场
            self._state.update({
                "in_pos": True,
                "dir": dir_now,
                "entry_bar": now_bar_idx,
                "best_z": z_now,    # 记录当下为最佳z的起点
            })
        elif was_in and dir_now == 0:
            # 已清仓
            self._state.update({
                "in_pos": False,
                "dir": 0,
```

```

        "entry_bar": None,
        "best_z": None,
    })
else:
    # 持仓持续：更新 best_z (“朝有利方向”的极值)
    if self._state["in_pos"] and z_now == z_now: # 非NaN
        if self._state["dir"] == +1:
            # 多价差：盈利来自 z 上升 (向 0 回归)
            self._state["best_z"] = max(self._state["best_z"], z_now)
        elif self._state["dir"] == -1:
            # 空价差：盈利来自 z 下降 (向 0 回归)
            self._state["best_z"] = min(self._state["best_z"], z_now)

def _maybe_trailing_stop(self, price_y: float, price_x: float, now_bar_idx: int, z_now: float):
    """若满足拖尾止盈条件，则平仓到0。"""
    if not self.p.use_trailing:
        return
    st = self._state
    if not st["in_pos"] or st["best_z"] is None or z_now != z_now:
        return
    # 持有期不足，不触发拖尾判断
    if self.p.trail_min_hold and st["entry_bar"] is not None:
        if (now_bar_idx - st["entry_bar"]) < self.p.trail_min_hold:
            return

    # 计算“对自己有利的最佳z”到当前z的回撤幅度
    hit = False
    if st["dir"] == +1:
        # 多价差：best_z - z_now > trail_z 则触发
        if (st["best_z"] - z_now) > self.p.trail_z:
            hit = True
    elif st["dir"] == -1:
        # 空价差：z_now - best_z > trail_z 则触发
        if (z_now - st["best_z"]) > self.p.trail_z:
            hit = True

    if hit:
        # 触发拖尾止盈：直接目标平仓
        self._rebalance_to_target(price_y, price_x, 0.0, 0.0)

```

3) 在 step() 中插入两行调用

在你原来的 step() 里（下单完成、杠杆检查之后、mark 之前/之后均可），加入状态更新与拖尾判定：

```
python
def step(self, t, price_y, price_x, alpha, beta, z_now):
    snap_before = self.exch.mark({self.y: price_y, self.x: price_x})
    tgt_y, tgt_x = self._target_pair_position(price_y, price_x, beta, z_now, snap_before["equity"])
    self._rebalance_to_target(price_y, price_x, tgt_y, tgt_x)
    self._enforce_leverage_cap(price_y, price_x)

    # === 新增: 更新状态、评估拖尾 ===
    # 1) 更新 best_z / 进出场状态
    bar_idx = len(self._equity_series) # 用已处理bar计数当索引
    self._on_position_change(bar_idx, z_now)
    # 2) 若满足拖尾触发, 立即清仓
    self._maybe_trailing_stop(price_y, price_x, bar_idx, z_now)

    snap = self.exch.mark({self.y: price_y, self.x: price_x})
    self._equity_series.append((t, snap["equity"]))
    self._snapshots.append(snap)
```

注: 这里把“bar 序号”用 `len(self._equity_series)` 代替 (简单可靠); 如果你更喜欢时间索引, 也可以传 `t` 并在状态里记录时间。

4) 最小示例 (直接能跑)

保持你原来的主程序, 只改策略参数, 让拖尾生效即可:

```
python
params = PairStatArbParams(
    lookback_beta=300,
    lookback_z=300,
    entry_z=1.5,
    exit_z=0.5,
    stop_z=3.5,
    capital_frac=0.2,
    max_leverage=5.0,
    use_maker=False,
    include_intercept=True,

    # 开启拖尾
    use_trailing=True,
    trail_z=0.8,          # 回撤超过 0.8 个 z 就落袋
    trail_min_hold=10,    # 至少持有 10 根 bar 才开始评估拖尾
)
```

小结 & 调参建议

- `trail_z` 越小越“紧”，越容易提前止盈；越大越“宽”，可能吃到更多回归但也容易回吐。
- 建议把 `exit_z` 保持较小（如 0.3—0.6），**拖尾**用于吃到“回归一半后的再延伸”；
- 若行情噪声大、`z` 抖动快，可适当**增大 `trail_min_hold`**（例如 20—50）避免“入场即抖出”。
- 如果你更喜欢**价格/收益百分比拖尾**（而不是 `z-score`），也很容易改：把 `best_z/trail_z` 换成“自进场以来的最高/最低权益”或“`spread` 的绝对变化/百分比”即可。

需要我帮你换成 **ATR/σ 的价格拖尾** 或者 **基于权益曲线的拖尾** 版本吗？

User:

单个高频因子的回测是否也可以用这个框架呢，可以的话，给个模板

ChatGPT:

可以。下面给你一个**“单个高频因子 → 单标的做多做空”**的回测模板，直接复用我们之前的 `SimpleExchange` 执行与结算层：

- 因子是**纯历史**（先算好因子列/或在模板里滚动标准化），
- 映射为**目标仓位**（连续或阈值式），
- 逐bar仿真下单、计费、估值（支持滑点/杠杆上限/冷却期等）。

把整段代码粘到一个 `.py` / Notebook 就能跑；把示例里的**合成因子**替换成你的**真实高频因子列**即可。

```
python
# ===== 依赖 =====
import numpy as np
import pandas as pd
from dataclasses import dataclass
from typing import Dict, Optional

# ===== 执行结算层: SimpleExchange (自包含版) =====
class SimpleExchange:
    def __init__(self, symbols, leverage=20.0, maker_fee=2e-4, taker_fee=5e-4, initial_balance=1000.0):
        self.symbols = list(symbols)
        self.leverage = float(leverage)
```

```
self.maker_fee = float(maker_fee)
self.taker_fee = float(taker_fee)
self.cash = float(initial_balance)
self.fees = 0.0
self.realized = 0.0
self.pos: Dict[str, float] = {s: 0.0 for s in self.symbols}
self.avg: Dict[str, float] = {s: 0.0 for s in self.symbols}
self.last: Dict[str, float] = {s: 0.0 for s in self.symbols}

def trade(self, symbol: str, side: int, price: float, qty: float, maker: bool=False):
    assert symbol in self.pos
    assert side in (1, -1)
    assert qty > 0 and price > 0
    fee_rate = self.maker_fee if maker else self.taker_fee
    fee = price * qty * fee_rate
    self.fees += fee
    self.cash -= fee
    p = self.pos[symbol]; a = self.avg[symbol]
    closing = min(abs(p), qty) if p * side < 0 else 0.0
    if closing > 0:
        realized = (price - a) * closing * (1.0 if p > 0 else -1.0)
        self.realized += realized
        self.cash += realized
        p = p + side * closing
    opening = qty - closing
    if opening > 0:
        if p == 0.0: a = price
        else:
            new_abs = abs(p) + opening
            a = (abs(p) * a + opening * price) / new_abs
        p = p + side * opening
    self.pos[symbol] = p; self.avg[symbol] = a; self.last[symbol] = price

def buy(self, symbol: str, price: float, qty: float, maker: bool=False): self.trade(symbol, +1, price, qty, maker)
def sell(self, symbol: str, price: float, qty: float, maker: bool=False): self.trade(symbol, -1, price, qty, maker)

def mark(self, prices: Dict[str, float]) -> Dict[str, float]:
    for s, px in prices.items():
        if s in self.last and px > 0: self.last[s] = px
    unreal = 0.0; notional = 0.0
    for s in self.symbols:
        if self.pos[s] != 0 and self.avg[s] > 0 and self.last[s] > 0:
            unreal += (self.last[s] - self.avg[s]) * self.pos[s]
        if self.last[s] > 0:
            notional += abs(self.pos[s]) * self.last[s]
    equity = self.cash + unreal
```

```

margin_req = notional / self.leverage if self.leverage > 0 else 0.0
used_leverage = (notional / equity) if equity > 0 else float("inf")
return {
    "cash": round(self.cash, 8), "fees": round(self.fees, 8),
    "realized_pnl": round(self.realized, 8),
    "unrealized_pnl": round(unreal, 8),
    "equity": round(equity, 8), "notional": round(notional, 8),
    "margin_required": round(margin_req, 8),
    "used_leverage": round(used_leverage, 6),
    "positions": {s: {"pos": self.pos[s], "avg": self.avg[s], "last": self.last[s]} for s in self.symbols},
}

```

===== 因子策略参数 =====

@dataclass

class FactorParams:

```

    z_window: int = 300          # 因子滚动标准化窗口 (z-score)
    entry_z: float = 0.5         # 死区阈值: |z| <= entry_z → 目标仓位=0 (降换手)
    z_cap: float = 2.5          # 逼近满仓的 z 尺度 (用于tanh映射/线性裁剪)
    capital_frac: float = 0.2    # 分配给该标的的名义额比例 (≈目标账户杠杆占比)
    max_leverage: float = 5.0    # 账户总杠杆上限 (超过则按比例缩仓)
    slip_bps: float = 1.0       # 简单滑点模型 (买: +bps; 卖: -bps)
    use_maker: bool = False      # 下单类型 (默认 taker)
    min_trade_qty: float = 1e-6  # 最小下单量
    cooldownBars: int = 0        # 信号翻转后冷却期 (避免抖动频繁换手)
    max_qty_step: Optional[float] = None # 每bar最大换手量 (限速), None表示不限

```

===== 单因子做多做空策略 =====

class SingleFactorStrategy:

"""

以单标的的价格 P_t 与因子 F_t (越大越看多; 可为任意HFT特征) 驱动仓位:

- 1) 对 F_t 做滚动标准化: $z_t = (F_t - \mu) / \sigma$
- 2) 映射到目标仓位因子 $s_t \in [-1, 1]$ (含死区与饱和):
 if $|z| \leq \text{entry_z} \rightarrow s=0$
 else $s = \tanh(z / z_cap)$
- 3) 目标名义额 = $\text{capital_frac} * \text{equity}$
 目标持仓量 $q^* = (s * \text{目标名义额}) / \text{price}$
- 4) 逐bar再平衡 (可含冷却/换手限速/杠杆上限/滑点)

"""

```

def __init__(self, exch: SimpleExchange, symbol: str, params: FactorParams):
    self.exch = exch
    self.s = symbol
    self.p = params
    self._equity_series = []
    self._snapshots = []

```

```

self._last_signal = 0 # 上次信号方向 (-1,0,1)
self._cooldown_left = 0

# --- 指标/信号 ---
@staticmethod
def _zscore(x: pd.Series, win: int) -> pd.Series:
    mu = x.rolling(win).mean()
    sd = x.rolling(win).std(ddof=0)
    return (x - mu) / sd

def _map_to_scale(self, z_now: float) -> float:
    """z → s ∈ [-1,1], 含死区与饱和 (tanh) """
    if np.isnan(z_now): return 0.0
    if abs(z_now) <= self.p.entry_z: return 0.0
    return float(np.tanh(z_now / max(self.p.z_cap, 1e-12)))

# --- 执行/风控 ---
def _apply_cooldown(self, s_now: float):
    # 信号方向翻转则进入冷却
    cur_dir = 0 if abs(s_now) < 1e-12 else (1 if s_now > 0 else -1)
    if self._cooldown_left > 0:
        self._cooldown_left -= 1
        return 0.0 # 冷却期内不持仓
    if self._last_signal != 0 and cur_dir != 0 and cur_dir != self._last_signal:
        self._cooldown_left = max(self.p.cooldownBars, 0)
        return 0.0
    self._last_signal = cur_dir
    return s_now

def _clip_turnover(self, tgt_qty: float, cur_qty: float) -> float:
    if self.p.max_qty_step is None: return tgt_qty
    step = tgt_qty - cur_qty
    lim = abs(self.p.max_qty_step)
    if step > lim: tgt_qty = cur_qty + lim
    if step < -lim: tgt_qty = cur_qty - lim
    return tgt_qty

def _exec_with_slippage(self, price: float, dqty: float):
    if abs(dqty) < self.p.min_trade_qty: return
    if dqty > 0:
        exec_px = price * (1.0 + self.p.slip_bps * 1e-4)
        self.exch.buy(self.s, exec_px, abs(dqty), maker=self.p.use_maker)
    else:
        exec_px = price * (1.0 - self.p.slip_bps * 1e-4)
        self.exch.sell(self.s, exec_px, abs(dqty), maker=self.p.use_maker)

```

```

def _enforce_leverage_cap(self, price: float):
    snap = self.exch.mark({self.s: price})
    if not np.isfinite(snap["used_leverage"]) or snap["used_leverage"] <= self.p.max_leverage:
        return
    scale = self.p.max_leverage / max(snap["used_leverage"], 1e-12)
    cur = self.exch.pos[self.s]
    tgt = cur * scale
    self._exec_with_slippage(price, tgt - cur)

# --- 主回测流程 ---
def run(self, df: pd.DataFrame, price_col: str, factor_col: str):
    """
    df: 含 price_col & factor_col, 索引为时间升序
    """
    assert price_col in df.columns and factor_col in df.columns
    P = df[price_col].astype(float)
    F = df[factor_col].astype(float)

    # 1) 因子标准化 (如你已有 z 列, 直接传入即可: 把下面注释掉, 改为 z=你的z列)
    z = self._zscore(F, self.p.z_window)

    for i, (t, price, z_now) in enumerate(zip(df.index, P.values, z.values)):
        # 2) z→目标规模 s ∈ [-1,1]
        s_now = self._map_to_scale(z_now)
        s_now = self._apply_cooldown(s_now)

        # 3) 目标名义额/目标仓位
        snap0 = self.exch.mark({self.s: price})
        alloc = max(self.p.capital_frac, 0.0) * max(snap0["equity"], 0.0)
        tgt_qty = 0.0 if alloc == 0 else (s_now * alloc) / max(price, 1e-12)
        # 换手限速
        cur_qty = self.exch.pos[self.s]
        tgt_qty = self._clip_turnover(tgt_qty, cur_qty)

        # 4) 再平衡 (带滑点/费率)
        self._exec_with_slippage(price, tgt_qty - cur_qty)

        # 5) 杠杆上限
        self._enforce_leverage_cap(price)

        # 6) 记账
        snap = self.exch.mark({self.s: price})
        self._equity_series.append((t, snap["equity"]))
        self._snapshots.append(snap)

```



```

    equity = pd.Series({t: v for t, v in self._equity_series})
    return equity, pd.DataFrame(self._snapshots, index=df.index)

# ===== 简易绩效 =====
def perf_stats(equity: pd.Series, bar_seconds: int = 60):
    equity = equity.dropna()
    ret = equity.pct_change().dropna()
    if len(ret) == 0:
        return {"sharpe": np.nan, "max_dd": np.nan, "return": np.nan, "vol": np.nan}
    per_year = int(round(365 * 24 * 3600 / bar_seconds))
    mu = ret.mean() * per_year
    sigma = ret.std(ddof=0) * np.sqrt(per_year)
    sharpe = mu / (sigma + 1e-12)
    cum = (1 + ret).cumprod()
    peak = cum.cummax()
    dd = (cum / peak - 1.0).min()
    total_return = equity.iloc[-1] / equity.iloc[0] - 1.0
    return {"sharpe": float(sharpe), "max_dd": float(dd), "return": float(total_return), "vol": float(sigma)}

# ===== 示例：合成一个“有预见性”的高频因子并回测 =====
if __name__ == "__main__":
    np.random.seed(0)
    n = 5000
    idx = pd.date_range("2025-01-01", periods=n, freq="S") # 1秒bar示例
    # 价格：弱随机游走
    ret = np.random.normal(0, 0.0005, size=n)
    price = 100 * np.exp(np.cumsum(ret))
    # 因子：与下一期收益有微弱相关 (toy示例：future ret 的有噪声前窥)
    future_ret = np.roll(ret, -1); future_ret[-1] = 0.0
    factor = future_ret + np.random.normal(0, 0.0005, size=n) # 你用真实高频因子替换它

    df = pd.DataFrame({"P": price, "F": factor}, index=idx)

    exch = SimpleExchange(symbols=["BTC"], leverage=10.0, taker_fee=5e-4, maker_fee=2e-4, initial_balance=10_000.0)
    params = FactorParams(
        z_window=300, entry_z=0.5, z_cap=2.0,
        capital_frac=0.3, max_leverage=3.0,
        slip_bps=1.0, use_maker=False,
        cooldown_bars=2, max_qty_step=None,
    )
    strat = SingleFactorStrategy(exch, symbol="BTC", params=params)

    equity, snaps = strat.run(df, price_col="P", factor_col="F")
    print("Performance:", perf_stats(equity, bar_seconds=1))

```

```
print("Final snapshot:", snaps.iloc[-1].to_dict())

# 可选画图
try:
    import matplotlib.pyplot as plt
    plt.figure(figsize=(8,3)); equity.plot(); plt.title("Equity Curve"); plt.tight_layout(); plt.show()
except Exception:
    pass
```

怎么把你的“高频因子”接进来？

- 1. 用你的数据构造 df，至少包含：
 - price_col (如 mid/last/close),
 - factor_col (你的因子值)。
 - 2. 如果你的因子已经**标准化**为 z，可以把 `z = ...` 直接替换掉；或把 `entry_z/z_cap` 改为**对原尺度的阈值/缩放**。
 - 3. 需要**离线先算**更复杂的特征（如 LOB 特征、微价格、OI、成交流等）时，把列算好塞进 df，框架只负责“映射→下单→结算”。
 - 4. 如果要**状态依赖**（如拖尾止损、冷却、分级加减仓），在这个模板里已经留了钩子：`cooldownBars`、`maxQtyStep`；也可以像我们上一条那样加“拖尾止盈”。
- 要我帮你把映射函数改成分位数分档（Q1..Q5 → 不同目标仓位）、或**风险平价版（名义额随波动/ATR自适应）**吗？

User:

好的，把映射函数改为分位数分档

ChatGPT:

好的！把“连续映射（tanh/z-score）”改成“**分位数分档**（quantile bins）”的**落地版模板**在这儿。思路：用滚动窗口计算因子的历史分位数阈值（避免未来函数，用 `.shift(1)`），把当前因子落到某一档，再映射为目标仓位权重 $s \in [-1, 1]$ $s \in [-1, 1]$ （支持把中间档设为 0 以降换手）。

下面是**可直接替换**的“单因子策略（分位数分档版）”。SimpleExchange 与 perf_stats 不变；只需把原来的 SingleFactorStrategy 换成这个 SingleFactorStrategyQ 即可。

```
python
from dataclasses import dataclass
import numpy as np
```

```

import pandas as pd
from typing import Dict, Optional, List

# === 执行层: 沿用你已有的 SimpleExchange (此处省略, 如果没有请贴回去) ===

@dataclass
class FactorParamsQ:
    # 分位数分档参数
    use_quantile_bins: bool = True
    n_bins: int = 5          # 档位数 (建议奇数: 3/5/7...)
    q_window: int = 2000     # 滚动分位数窗口 (避免未来函数会 shift(1))
    bin_weights: Optional[List[float]] = None # 每档的目标权重s∈[-1,1], 长度==n_bins; None则自动等间距
    neutral_bin: Optional[int] = None        # 指定某档权重=0 (如中位档)

    # 执行/风控 (与之前一致)
    capital_frac: float = 0.2
    max_leverage: float = 5.0
    slip_bps: float = 1.0
    use_maker: bool = False
    min_trade_qty: float = 1e-6
    cooldownBars: int = 0
    max_qty_step: Optional[float] = None

class SingleFactorStrategyQ:
    """
    单标的 + 单因子 (高频可用):
    - 因子 F_t 用滚动窗口 q_window 计算分位数阈值 (等概率切分成 n_bins 档)
    - 当前 F_t 落入第 k 档 → 目标仓位权重 s = bin_weights[k] (默认等间距映射到 [-1,1])
    - 名义额 = capital_frac * equity; 目标持仓 = s * 名义额 / 价格
    - 支持冷却/换手限速/杠杆上限/滑点
    """
    def __init__(self, exch, symbol: str, params: FactorParamsQ):
        self.exch = exch
        self.s = symbol
        self.p = params
        self._equity_series = []
        self._snapshots = []
        self._last_dir = 0
        self._cooldown_left = 0

        # 准备权重
        if self.p.bin_weights is None:
            # 等间距 [-1,1]
            w = np.linspace(-1.0, 1.0, self.p.n_bins).tolist()

```

```

# 可选: 把中间档设为 0 (推荐奇数档位)
if self.p.neutral_bin is None and self.p.n_bins % 2 == 1:
    mid = self.p.n_bins // 2
    w[mid] = 0.0
    self.p.neutral_bin = mid
elif self.p.neutral_bin is not None and 0 <= self.p.neutral_bin < self.p.n_bins:
    w[self.p.neutral_bin] = 0.0
self.p.bin_weights = w
else:
    assert len(self.p.bin_weights) == self.p.n_bins, "bin_weights 长度必须等于 n_bins"

# ----- 工具函数 -----
def _prepare_thresholds(self, F: pd.Series) -> pd.DataFrame:
    """
    返回一个 DataFrame, 每列是一个滚动分位数阈值 (p=1/n, 2/n, ..., (n-1)/n), 并向后 shift(1) 防未来函数。
    """
    probs = [(i / self.p.n_bins) for i in range(1, self.p.n_bins)]
    th_dict = {}
    # 注意: rolling.quantile 是包含当前点的, 故必须 shift(1)
    for p in probs:
        th = F.rolling(self.p.q_window, min_periods=self.p.q_window).quantile(p)
        th_dict[p] = th.shift(1)
    th_df = pd.DataFrame(th_dict, index=F.index)
    return th_df

def _bin_index(self, f_now: float, th_row: pd.Series) -> Optional[int]:
    """
    给定当期因子值和该期的分位数阈值行, 返回档位索引 0..n_bins-1。
    若阈值存在 NaN (窗口未满), 返回 None 表示暂不交易。
    """
    vals = th_row.values
    if np.isnan(vals).any():
        return None
    # thresholds 升序; searchsorted 右侧插入点, 即有多少个阈值 <= f_now
    k = int(np.searchsorted(vals, f_now, side="right"))
    # 保证边界
    if k < 0: k = 0
    if k >= self.p.n_bins: k = self.p.n_bins - 1
    return k

def _apply_cooldown(self, s_now: float) -> float:
    cur_dir = 0 if abs(s_now) < 1e-12 else (1 if s_now > 0 else -1)
    if self._cooldown_left > 0:
        self._cooldown_left -= 1
    return 0.0

```

```

    if self._last_dir != 0 and cur_dir != 0 and cur_dir != self._last_dir:
        self._cooldown_left = max(self.p.cooldownBars, 0)
        return 0.0
    self._last_dir = cur_dir
    return s_now

def _clip_turnover(self, tgt_qty: float, cur_qty: float) -> float:
    if self.p.max_qty_step is None:
        return tgt_qty
    step = tgt_qty - cur_qty
    lim = abs(self.p.max_qty_step)
    if step > lim: tgt_qty = cur_qty + lim
    if step < -lim: tgt_qty = cur_qty - lim
    return tgt_qty

def _exec_with_slippage(self, price: float, dqty: float):
    if abs(dqty) < self.p.min_trade_qty:
        return
    if dqty > 0:
        px = price * (1.0 + self.p.slip_bps * 1e-4)
        self.exch.buy(self.s, px, abs(dqty), maker=self.p.use_maker)
    else:
        px = price * (1.0 - self.p.slip_bps * 1e-4)
        self.exch.sell(self.s, px, abs(dqty), maker=self.p.use_maker)

def _enforce_leverage_cap(self, price: float):
    snap = self.exch.mark({self.s: price})
    if not np.isfinite(snap["used_leverage"]) or snap["used_leverage"] <= self.p.max_leverage:
        return
    scale = self.p.max_leverage / max(snap["used_leverage"], 1e-12)
    cur = self.exch.pos[self.s]
    tgt = cur * scale
    self._exec_with_slippage(price, tgt - cur)

# ----- 主流程 -----
def run(self, df: pd.DataFrame, price_col: str, factor_col: str):
    """
    df: 包含 price_col 与 factor_col, 索引为时间升序
    """
    assert price_col in df.columns and factor_col in df.columns
    P = df[price_col].astype(float)
    F = df[factor_col].astype(float)

    th_df = self._prepare_thresholds(F) # 每期的分位数阈值 (已shift防未来)

    for t, price in zip(df.index, P.values):

```

```

f_now = F.at[t]
th_row = th_df.loc[t]
k = self._bin_index(f_now, th_row)
if k is None:
    s_now = 0.0 # 窗口未满, 先不交易
else:
    s_now = float(self.p.bin_weights[k])

# 冷却期
s_now = self._apply_cooldown(s_now)

snap0 = self.exch.mark({self.s: price})
alloc = max(self.p.capital_frac, 0.0) * max(snap0["equity"], 0.0)
tgt_qty = 0.0 if alloc == 0 else (s_now * alloc) / max(price, 1e-12)

cur_qty = self.exch.pos[self.s]
tgt_qty = self._clip_turnover(tgt_qty, cur_qty)

self._exec_with_slippage(price, tgt_qty - cur_qty)
self._enforce_leverage_cap(price)

snap = self.exch.mark({self.s: price})
self._equity_series.append((t, snap["equity"]))
self._snapshots.append(snap)

equity = pd.Series({t: v for t, v in self._equity_series})
return equity, pd.DataFrame(self._snapshots, index=df.index)

```

怎么用（与原模板一致，只换类/参数）

```

python
# 准备 df: 至少包含 "P" (价格) 与 "F" (你的高频因子)
# exch = SimpleExchange(symbols=["BTC"], ...)

params_q = FactorParamsQ(
    n_bins=5,          # 5档: 强空/弱空/中性/弱多/强多
    q_window=2000,     # 例如秒级数据可取 2k~10k; 分频越高, 窗口可更大
    bin_weights=None,  # None→自动等间距[-1,1]且中档置0; 或自定义如[-1,-0.3,0,0.3,1]
    neutral_bin=None,  # 传 None 会在奇数档时自动把中档置0
    capital_frac=0.2,
    max_leverage=3.0,
    slip_bps=1.0,
    cooldownBars=2,    # 档位翻转后2根bar冷静期
)

```

```
strat = SingleFactorStrategyQ(exch, symbol="BTC", params=params_q)
equity, snaps = strat.run(df, price_col="P", factor_col="F")
```

关键细节（避免“未来函数” & 过拟合）

- 滚动分位数用 **.shift(1)**：阈值只基于“到上一根为止”的历史，当前点仅用于落档，不参与阈值估计。
- 分档数量 **n_bins**：越多越精细，但切换更频繁；常用 3/5/7 档。
- 中性档置 0：显著降低换手（特别是信号在中位附近抖动的因子）。
- **q_window**：越大越稳、越抗噪；但对“因子分布漂移”的适应更慢。
- **bin_weights**：可自定义阶梯（例如 `[-1, -0.5, 0, 0.5, 1]` 或偏保守 `[-1, -0.25, 0, 0.25, 1]`）。

如果你愿意，我可以再给一个**“分位数×波动归一化”的风险平价版**（让名义额随近期波动/ATR 自适应），或多标的批量跑的并行骨架。
