重构代码笔记

参考《重构》第二版改进我的交易系统代码。
主要目标是system.py, tradesys.py, monitoring.py三个文件。
先建立一个分支:git branch refactor
然后进入该分支:git checkout refactor
开始干活。
所谓重构是这样一个过程：“在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构”。本质上说，重构就是“在代码写好之后改进它的设计”。
重构过程的精髓所在：小步修改，每次修改后就运行测试。
每次修改都在本地提交，提交到远程前，合并零碎的提交。
好代码的检验标准就是人们是否能轻而易举地修改它。
事不过三，三则重构。

代码的坏味道

1. 神秘命名(Mysterious Name）
   改变函数声明，变量改名，字段改名。
2. 重复代码（Duplicated Code）
   提炼函数，移动语句，函数上移。
3. 过长函数（Long Function）关键在于“做什么”和“如何做”之间的语义距离。其之前有注释的语句往往可以独立成函数。
   提炼函数，查询取代临时变量，引入参数对象，保持对象完整，以命令取代函数，分解条件表达式，以多态取代条件表达式，拆分循环。
4. 过长参数列表（Long Parameter List）以查询取代参数，保持对象完整，引入参数对象，移除标记参数，函数组合成类。
5. 全局数据（Global Data）封装变量。
6. 可变数据（Mutable Data）封装变量，拆分变量，移动语句，提炼函数，查询函数与修改函数分离，移除设置函数，以查询取代派生变量，函数组合成类，函数组合成变换，将引用对象改为值对象。
7. 发散式变化（Divergent Change）每次只关心一个上下文。拆分阶段、搬移函数、提炼函数、提炼类。
8. 霰弹式修改（Shotgun Surgery）搬移函数，搬移字段、函数组合成类、函数组合成变换、拆分阶段、内联函数、内联类。
9. 依恋情结（Feature Envy）两个模块中的函数交流频繁。搬移函数、提炼函数。
10. 数据泥团（Data Clumps）数据项总在一起出现。提炼类、引入参数对象、保持对象完整。
11. 基本类型偏执（Primitive Obsession）不要只用基本类型。以对象取代基本类型、以子类取代类型码、以多态取代条件表达式。提炼类、引入参数对象。
12. 重复的switch （Repeated Switches）以多态取代表达式。
13. 循环语句（Loops）以管道取代循环。

14. 冗赘的元素（Lazy Element）内联函数、内联类、折叠继承体系。
15. 夸夸其谈通用性（Speculative Generality）折叠继承体系、内联函数、内联类、改变函数声明。如果函数或类的唯一用户是测试用例，这就飘出了坏味道“夸夸其谈通用性”。移除死代码。
16. 临时字段（Temporary Field）提炼类、搬移函数、引入特例。
17. 过长的消息链（Message Chains）不同对象依次引用对方，隐藏委托关系、提炼函数、搬移函数。
18. 中间人（Middle Man）过度委托。移除中间人，内联函数，以委托取代超类，以委托取代子类。
19. 内幕交易（Insider Trading） 搬移函数、搬移字段、隐藏委托关系、以委托取代子类、以委托取代超类。
20. 过大的类（Large Class）提炼类、提炼超类、以子类取代类型码、提炼类、提炼超类、以子类取代类型码。
21. 异曲同工的类（Alternative Classes with DifferentInterfaces）改变函数声明、搬移函数、提炼超类。
22. 纯数据类（Data Class）封装记录、移除设值函数、搬移函数、提炼函数。
23. 被拒绝的遗赠（Refused Bequest）函数下移、字段下移、以委托取代子类、以委托取代超类。
24. 注释（Comments）提炼函数、改变函数声明、引入断言

接下来，先找我代码里的“坏味道”吧，以system.py为例。

1. 神秘命名
   变量名
   81行 cash0 => cash_init
   90-91行 td, tp => hold_days, trade_profit
   115行 bd => buy_days
   252行 retrate => profit_rate
   253行 ret => returns
   260行 bench
   函数名
   127行 def test => backtest
   317行 def riskAnaly => risk_analyse
   370行 def draw => drawing_process
   重构方法:
   变量改名（Rename Variable）
   改变函数声明(Change Function Declaration)
   函数名，变量名等用小写加_。类名首字母大写，驼峰命名法。全局变量全部大写。
2. 过长函数（Long Function）
   main()函数和backtest()函数，尤其后者。
   重构方法
   提炼函数，查询取代临时变量，引入参数对象，保持对象完整，以命令取代函数，分解条件表达式，以多态取代条件表达式，拆分循环。
   提炼函数
   “将意图与实现分开”：如果你需要花时间浏览一段代码才能弄清它到底在干什么，那么就应该将其提

炼到一个函数中，并根据它所做的事为其命名。以后再读到这段代码时，你一眼就能看到函数的用途，大多数时候根本不需要关心函数如何达成其用途（这是函数体内干的事）。

3. 全局变量 （Global Data）
   重构方法: 封装变量
   不可变性是强大的代码防腐剂。
   全局变量，回测用的
   cash = 1000000 # 初始资金
   cash_init = 1000000
   commit_rate = 0.0006 # 佣金税收费率
   stocks = 0 # 持仓量
   cost = 0.0 # 交易成本
   value = 0.0 # 股票市值
   profit = 0.0 # 净收益
   buy_value = 0.0 # 买入后的总资产
   sell_value = 0.0 # 卖出后的总资产
   hold_days = [] # 持股天数
   trade_profit = [] # 每次交易的收益(为负则是亏损)
4. 数据泥团（Data Clumps）
   重构方法: 提炼类、引入参数对象、保持对象完整。
   如果某些数据和某些函数总是一起出现，某些数据经常同时变化甚至彼此相依，这就表示你应该将它们分离出去。一个有用的测试就是问你自己，如果你搬移了某些字段和函数，会发生什么事？其他字段和函数是否因此变得无意义？
   函数组合成类:如果发现一组函数形影不离地操作同一块数据（通常是将这块数据作为参数传递给函数），我就认为，是时候组建一个类了。
   提炼Trade类。提炼经验:如果某些函数频繁调用某个类的成员变量，就可以考虑把它们包含到该类中。

代码重构暂时先到这儿，下面看设计模式。主要目标是把具体策略跟回测过程等分离开，这样改变策略就不用动程序的其它部分了。
python设计模式实现: https://github.com/faif/python-patterns
先看老祖宗:《设计模式:可复用面向对象软件的基础》
每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动。一个模式包含四个基本要素:①模式名称(pattern name)；②问题(problem)；③解决方案(solution)；④效果(consequences)。
选择设计模式:提出问题，看设计模式的意图、目的等，看二者或设计模式的组合是否与问题匹配，关注自己设计的可变部分。
设计模式的目的主要分为创建、结构和行为三种。

表1-2　设计模式所支持的设计的可变方面

目　的	设计模式	可变的方面
创建	Abstract Factory(3.1)	产品对象家族
	Builder(3.2)	如何创建一个组合对象
	Factory Method(3.3)	被实例化的子类
	Prototype(3.4)	被实例化的类
	Singleton(3.5)	一个类的唯一实例
结构	Adapter(4.1)	对象的接口
	Bridge(4.2)	对象的实现
	Composite(4.3)	一个对象的结构和组成
	Decorator(4.4)	对象的职责，不生成子类
	Facade(4.5)	一个子系统的接口
	Flyweight(4.6)	对象的存储开销
	Proxy(4.7)	如何访问一个对象；该对象的位置
行为	Chain of Responsibility(5.1)	满足一个请求的对象
	Command(5.2)	何时、怎样满足一个请求
	Interpreter(5.3)	一个语言的文法及解释
	Iterator(5.4)	如何遍历、访问一个聚合的各元素
	Mediator(5.5)	对象间怎样交互、和谁交互
	Memento(5.6)	一个对象中哪些私有信息存放在该对象之外，以及在什么时候进行存储
	Observer(5.7)	多个对象依赖于另外一个对象，而这些对象又如何保持一致
	State(5.8)	对象的状态
	Strategy(5.9)	算法
	Template Method(5.10)	算法中的某些步骤
	Visitor(5.11)	某些可作用于一个（组）对象上的操作，但不修改这些对象的类

我的问题，似乎应该使用策略模式(strategy)。

一个设计模式只有当它提供的灵活性是真正需要的时候，才有必要使用。

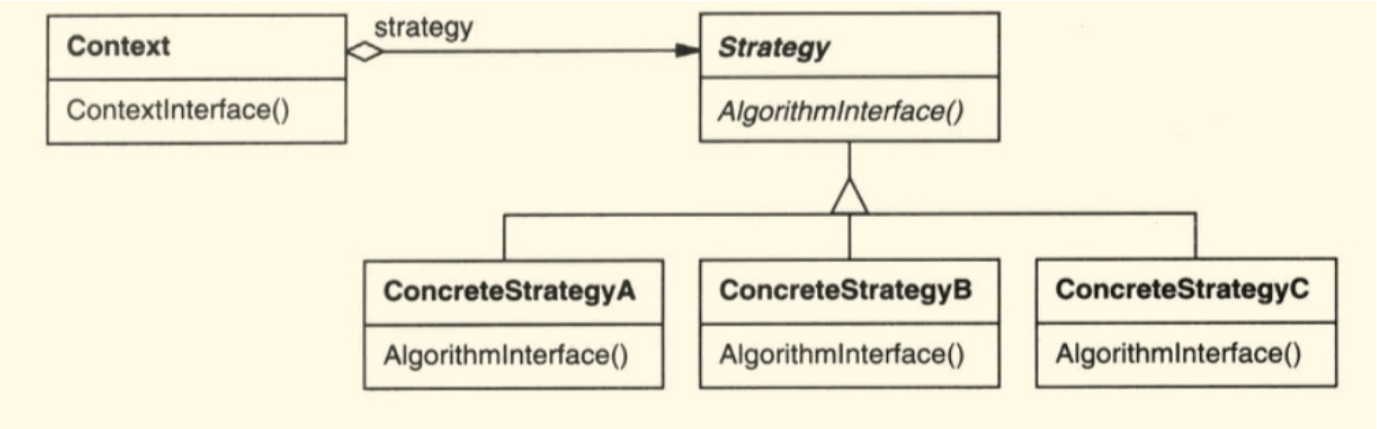
直接跳到策略模式吧。

①意图:定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

②动机:实现一个功能有并列的不同的算法，可能相互替换。如果与客户程序融合在一起，难以修改和维护。

③适用性:许多相关类仅仅是行为有异；一个算法的不同变体；算法使用客户不应知道的数据；一个类定义了多种行为。

④结构



context与抽象策略类交互，提供数据，客户创建并传递一个抽象策略类给context，随后只与context交互。含有许多条件语句的代码通常意味着需要使用策略模式。
⑤缺点:就是一个客户要选择一个合适的策略就必须知道这些策略到底有何不同。此时可能不得不向客户暴露具体的实现问题。因此仅当这些不同行为变体与客户相关的行为时，才需要使用策略模式。
⑥实现:参考其它书或资料吧。

参考其它书
《设计模式之禅(第2版)》 秦小波 著
要拥抱变化，使用设计模式。
策略模式:父类定义接口，子类实现接口。
用户实例化具体策略，通过构造函数传递给上下文对象。
缺点:必须向上层模块暴露策略实现，可以使用其他模式来修正这个缺陷，如工厂方法模式、代理模式或享元模式。
如果策略数量超过4个，考虑使用混合模式。
策略模式的意图是封装算法，它认为“算法”已经是一个完整的、不可拆分的原子业务（注意这里是原子业务，而不是原子对象），即其意图是让这些算法独立，并且可以相互替换，让行为的变化独立于拥有行为的客户。
策略模式关注的是算法替换的问题，一个新的算法投产，旧算法退休，或者提供多种算法由调用者自己选择使用，算法的自由更替是它实现的要点。换句话说，策略模式关注的是算法的完整性、封装性，只有具备了这两个条件才能保证其可以自由切换。
策略模式封装的是不同的算法，算法之间没有交互，以达到算法可以自由切换的目的。
工厂方法模式+策略模式混编。

《大话设计模式》 吴强著
使用策略模式的场景:1、系统有许多类，而他们的区别仅仅在于它们的行为。2、动态选择几种算法中的一种。3、一个对象有很多行为。
步骤:1、定义抽象角色类，定义好各个实现的共同抽象方法。2、定义具体策略类，具体实现父类的共同方法。3、定义环境角色类，私有化申明抽象角色变量,重载构造方法,执行抽象方法。

极客时间《设计模式之美》专栏 王争
继承会产生组合爆炸的问题。继承结构稳定，层数不多，可以用继承。否则用组合。
设计原则:单一职责；对扩展开放，对修改封闭；里氏替换(子类对象能替换父类对象出现的地方)；接口隔离原则；依赖反转原则(抽象不要依赖实现细节，实现细节要依赖抽象)；避免重复原则；迪米特法则(高内聚、低耦合，最少知识原理)。

依赖注入:不在类内部新建类，而是通过构造函数的参数传入对象。
重构的时机:持续重构。
策略模式，最常见的应用场景是，利用它来避免冗长的 if-else 或 switch 分支判断。其优势是在运行时动态确定使用的算法。
设计原则和思想其实比设计模式更加普适和重要，掌握了代码的设计原则和思想，我们甚至可以自己创造出来新的设计模式。
策略模式第一步:将策略定义分离出来。

设计原则和思想是心法，设计模式只是招式。掌握心法，以不变应万变，无招胜有招。所以，设计原则和思想比设计模式更加普适、重要。掌握了设计原则和思想，我们能更清楚地了解为什么要用某种设计模式，就能更恰到好处地应用设计模式，甚至我们还可以自己创造出来新的设计模式。
设计是先有问题，后有方案。

现在，来看用python具体实现设计模式。
参考:
《人人都懂设计模式》 罗伟富 著
python类的定义。_name为私有变量，__name为保护变量。类的继承:class 子类(基类): 调用基类的成员函数使用super()前缀。构造函数__init__()，析构函数__del__()，一般类的metaclass是type。
《精通python设计模式》 [荷]Sakis Kasampalis 著
设计模式是在已有的方案之上发现更好的方案（而不是全新发明）。若你一个方案都没有，又何谈找一

个更好的呢？先行动起来，用你的技能尽可能漂亮地解决问题。若代码评审者没有反对意见，而你经过一段时间后还是觉得自己的方案足够漂亮灵活，那也就意味着没必要浪费时间纠结使用哪种模式。你也许还能发现更好的设计模式。谁知道呢，关键是不要为了强迫自己使用已有的设计模式而限制了你的创造力。随处使用设计模式与过早优化一样，都是误入歧途。

下面尝试重构我的交易回测代码:

```
# 实际进行回测过程
def test_process(data_day, data_week, trade):
    days, days_s_slop, days_l_slop, days_close, macd, dif, dea =
make_temp_data(data_day, data_week)
    # 回测指标数据
    total_cash = [] # 总现金
    total_stock = [] # 总持股数
    total_value = [] # 总持仓市值
    total_profit = [] # 总盈利
    total_cost = [] # 总成本

    i = 0
    bIn = False
    bTrade = False
    buy_days = -1 # 买入时的索引
    buy_price = 0.0 # 买入价格
    stop_loss = 0.05 # 买入止损比例5%
    stop_profit = 0.1 # 止盈比例10%
    highest_price = 0.0 # 交易时的最高价
    stoptimes = 0 # 止损止盈卖出次数
    bStop = False # 是否止盈止损
    for day in days:
        # print("bug测试", cash, cash_init)
        # 进行交易
        if bTrade == True and bIn == True:
            trade.do_trade("buy", days_close[i])
            buy_days = i
            buy_price = days_close[i]
            highest_price = buy_price
            bTrade = False
        elif bTrade == True and bIn == False:
            trade.do_trade("sell", days_close[i], hold_days = i - buy_days + 1, bStop
= bStop)
            # trade.append_hold_days(i - buy_days + 1)
            buy_days = -1
            highest_price = 0.0
```

```
            bTrade = False
            bStop = False

        trade.make_record(days_close[i])
        week = data_week[data_week.日期 <= day]
        if days_close[i] > highest_price:
            highest_price = days_close[i]
        if len(week) == 0:
            i += 1
            continue
        else:
            week = week.iloc[-1, :]
            # print(day, week)
        judge = week["短期均线斜率"] > 0 and week["长期均线斜率"] > 0 and week["短期均线"] > week["长期均线"]
        if judge and bIn == False:
           if i >= 1:
               if macd[i] > 0 and macd[i-1] < 0 and dif[i-1] < dea[i-1] and dif[i] > dea[i]:
                   # print("买点", day)
                   bIn = True
                   bTrade = True
                   continue

        if bIn == True:
            # 判断是否到达止损止盈点
            # 低于买入价，按买入止损
            if days_close[i] < buy_price:
                if (buy_price - days_close[i])/buy_price >= stop_loss:
                    bStop = True
                    stoptimes += 1
                    bIn = False
                    bTrade = True
                    bStop = True
                    continue
            else: # 高于买入价，按浮盈止盈
                if (highest_price - days_close[i])/highest_price >= stop_profit:
                    bStop = True
                    stoptimes += 1
                    bIn = False
                    bTrade = True
                    bStop = True
                    continue
```

```
            # 判断是否达到出场条件
            if i > 2:
                if (macd[i-2] < 0.0 and macd[i-1] < 0.0 and macd[i] < 0.0):
                    # print("卖点", day
                    bIn = False
                    bTrade = True
        i += 1
```

这个函数太长了，回测过程跟具体策略算法纠缠在一起，要使用新的策略很困难。打算采用策略模式进行重构，然而粗看起来怎么写仍然很难。先把大函数拆分成小函数吧。

```
# 临时定义全局变量
bIn = False
bTrade = False
buy_days = -1 # 买入时的索引
buy_price = 0.0 # 买入价格
stop_loss = 0.05 # 买入止损比例5%
stop_profit = 0.1 # 止盈比例10%
highest_price = 0.0 # 交易时的最高价
bStop = False # 是否止盈止损


# 具体交易过程
def doTrade(trade, i):
    global bTrade, bIn, days_close, highest_price, bStop, buy_days
    if bTrade == True and bIn == True:
        trade.do_trade("buy", days_close[i])
        buy_days = i
        buy_price = days_close[i]
        highest_price = buy_price
        bTrade = False
    elif bTrade == True and bIn == False:
        trade.do_trade("sell", days_close[i], hold_days = i - buy_days + 1, bStop =
bStop)
        # trade.append_hold_days(i - buy_days + 1)
        buy_days = -1
        highest_price = 0.0
        bTrade = False
        bStop = False


# 记录最高价
def record_highest_price(price):
    global highest_price
```

```
    if price > highest_price:
        highest_price = price


# 第一层滤网，周线层面上判断
def level1(week):
    judge = week["短期均线斜率"] > 0 and week["长期均线斜率"] > 0 and week["短期均线"] > week["长期均线"]
    return judge


# 第二层滤网，日线层面上判断
def level2(macd, dif, dea, i):
    return macd[i] > 0 and macd[i-1] < 0 and dif[i-1] < dea[i-1] and dif[i] > dea[i]


# 止损
def b_stop_loss(price):
    global buy_price, stop_loss
    return (buy_price - price)/buy_price >= stop_loss


# 止盈
def b_stop_profit(price):
    global highest_price, stop_profit
    return (highest_price - price)/highest_price >= stop_profit


# 出场
def b_out(macd, i):
    return (macd[i-2] < 0.0 and macd[i-1] < 0.0 and macd[i] < 0.0)


# 具体策略
def step(i, price, week, macd, dif, dea):
    global bTrade, bIn, days_close, highest_price, bStop, stop_loss, stop_profit
    judge = level1(week)
    if judge and bIn == False:
       if i >= 1:
           if level2(macd, dif, dea, i):
               # print("买点", day)
               bIn = True
               bTrade = True
```

```
                return

    if bIn == True:
        # 判断是否到达止损止盈点
        # 低于买入价，按买入止损
        if price < buy_price:
            if b_stop_loss(price):
                bStop = True
                bIn = False
                bTrade = True
                return
        else: # 高于买入价，按浮盈止盈
            if b_stop_profit(price):
                bStop = True
                bIn = False
                bTrade = True
                return
        # 判断是否达到出场条件
        if i > 2:
            if b_out(macd, i):
                # print("卖点", day
                bIn = False
                bTrade = True


# 实际进行回测过程
def test_process(data_day, data_week, trade):
    global bTrade, bIn, days_close, highest_price, bStop, stop_loss, stop_profit
    days, days_s_slop, days_l_slop, days_close, macd, dif, dea =
make_temp_data(data_day, data_week)
    i = 0

    for day in days:
        # print("bug测试", cash, cash_init)
        # 进行交易
        doTrade(trade, i)

        trade.make_record(days_close[i])
        week = data_week[data_week.日期 <= day]
        record_highest_price(days_close[i])
        if len(week) == 0:
            i += 1
            continue
```

```
        else:
            week = week.iloc[-1, :]
            # print(day, week)
        step(i, days_close[i], week, macd, dif, dea)
        i += 1


# 生成临时数据
def make_temp_data(data_day, data_week):
    days = data_day["日期"].values
    days_s_slop = data_day["短期均线斜率"].values
    days_l_slop = data_day["长期均线斜率"].values
    days_close = data_day["收盘"].values
    macd = data_day["macd"].values
    dif = data_day["dif"].values
    dea = data_day["dea"].values
    return days, days_s_slop, days_l_slop, days_close, macd, dif, dea
```

现在好多了，但使用了全局变量。再提炼为类吧

```
# 封装回测过程
class Trade:
    def __init__(self):
        self._bIn = False
        self._bTrade = False
        self._buy_days = -1 # 买入时的索引
        self._buy_price = 0.0 # 买入价格
        self._stop_loss = 0.05 # 买入止损比例5%
        self._stop_profit = 0.1 # 止盈比例10%
        self._highest_price = 0.0 # 交易时的最高价
        self._bStop = False # 是否止盈止损


    # 具体交易过程
    def doTrade(self, broker, i, price):
        if self._bTrade == True and self._bIn == True:
            broker.do_trade("buy", price)
            self._buy_days = i
            self._buy_price = price
            self._highest_price = self._buy_price
            self._bTrade = False
        elif self._bTrade == True and self._bIn == False:
            broker.do_trade("sell", price, hold_days = i - self._buy_days + 1, bStop
= self._bStop)
```

```
            # trade.append_hold_days(i - buy_days + 1)
            self._buy_days = -1
            self._highest_price = 0.0
            self._bTrade = False
            self._bStop = False


    # 记录最高价
    def record_highest_price(self, price):
        if price > self._highest_price:
            self._highest_price = price


    # 第一层滤网，周线层面上判断
    def level1(self, week):
        judge = week["短期均线斜率"] > 0 and week["长期均线斜率"] > 0 and week["短期均线"] > week["长期均线"]
        return judge


    # 第二层滤网，日线层面上判断
    def level2(self, macd, dif, dea, i):
        return macd[i] > 0 and macd[i-1] < 0 and dif[i-1] < dea[i-1] and dif[i] > dea[i]


    # 止损
    def b_stop_loss(self, price):
        return (self._buy_price - price)/self._buy_price >= self._stop_loss


    # 止盈
    def b_stop_profit(self, price):
        return (self._highest_price - price)/self._highest_price >= self._stop_profit


    # 出场
    def b_out(self, macd, i):
        return (macd[i-2] < 0.0 and macd[i-1] < 0.0 and macd[i] < 0.0)


    # 具体策略
    def step(self, i, price, week, macd, dif, dea):
```

```
        judge = self.level1(week)
        if judge and self._bIn == False:
            if i >= 1:
                if self.level2(macd, dif, dea, i):
                    # print("买点", day)
                    self._bIn = True
                    self._bTrade = True
                    return

        if self._bIn == True:
        # 判断是否到达止损止盈点
            # 低于买入价，按买入止损
            if price < self._buy_price:
                if self.b_stop_loss(price):
                    self._bStop = True
                    self._bIn = False
                    self._bTrade = True
                    return
            else: # 高于买入价，按浮盈止盈
                if self.b_stop_profit(price):
                    self._bStop = True
                    self._bIn = False
                    self._bTrade = True
                    return
            # 判断是否达到出场条件
            if i > 2:
                if self.b_out(macd, i):
                    # print("卖点", day
                    self._bIn = False
                    self._bTrade = True


# 实际进行回测过程
def test_process(data_day, data_week, broker):
    days, days_s_slop, days_l_slop, days_close, macd, dif, dea =
make_temp_data(data_day, data_week)
    i = 0
    trade = Trade()

    for day in days:
        # print("bug测试", cash, cash_init)
        # 进行交易
        trade.doTrade(broker, i, days_close[i])
```

```
        broker.make_record(days_close[i])
        week = data_week[data_week.日期 <= day]
        trade.record_highest_price(days_close[i])
        if len(week) == 0:
            i += 1
            continue
        else:
            week = week.iloc[-1, :]
            # print(day, week)
        trade.step(i, days_close[i], week, macd, dif, dea)
        i += 1
```

接下来应该将策略的可变部分分离出来，用策略模式，这样就可以提高程序的可复用性。但是之前写的程序这些部分是交结在一起的，另外感觉越改越像backtrader框架。我这水平，还是做个使用者好了。