# 导语

目前交易系统来以SQL/Oracle为核心,通信方式采用同步一问一答一等待、报盘等核心节点单点等方式,极大的影响了客户的体验感以及交易效率。

对于机构技术部来说,每天需要投入大量的人力去做重复的重启、请磁盘、定时查询核心节点软件是否正常市场需要一款具有稳定可靠、低延迟、有较高并发能力、支持分多节点部署、支持数据持久化的的高效内存交易系统

# 基础篇

## 交易系统位置图



## 现状以及难点

* 交易系统本身是无法独立存在于行业体系的,每一个交易系统时刻与各种系统进行着数据的交互，集中交易柜台、PB系统、内控、TA、估值甚至OA系统。而这些系统架构有新有旧、功能繁杂。

尤其是集中交易系统设计老旧,新系统对接老系统时需要充分考虑丢包、超时、谨慎重传。

* 整个行业并无统一的通信协议标准,目前各大厂商几乎都是各自为战,相互协议并不互通。大体上均为以key-value为核心的明文文本协议。
* 目前交易所业务种类繁多,而且交易所并不公布长远的业务规划,导致系统设计时需要充分考虑现有的业务兼容同时也需要有一定的扩展性
* 业务需求难以整合,各家基本上都有各家自己的玩法,业务需求的抽象难度比较高
* 券商运维能力低下,整个行业的运维模式很大程度上是机构技术部门无法适应命令行、问题运维基本上由IT供应商驻点完成。交易系统设计时需要考虑将需要认为干预的点降低到最小。
* 交易系统运行环境对于技术人员来说是不可达的,技术人员无权限访问后台甚至无权限登录客户端。往往出问题时,技术人员只能通过客户对界面的反馈做出问题的判断,这中间经常会因为客户不耐烦或者表述造成大量的信息失真,影响交易人员的判断。
* 交易系统中流量并不是一个平缓的曲线,往往随着市场的剧烈波动产生巨大的数据波动。对交易系统而言最考验交易系统的时候就是市场大牛市或者大幅度跳水。对交易系统来说瞬时处理能力至关重要,往往瞬时处理时风控功能会导致大量的数据抢占情况。

一套交易系统用户数量并不会太多,在信托行业中融信托因为杠杆门槛较低,所以客户数量也偏多,即便如此,巅峰时点用户在线数量也仅仅是4000多。

* 数据的准确性至关重要,涉及到财务报表,对于交易系统而言,数据在精度范围内必须严格准确。资产负债差一分钱会计上就会被认定为资产负债不平衡导致整表无法正常使用。

## 交易系统设计准则

目前硬件发展迅速,一台顶配HP580报价基本上在20-50W之间。对于交易系统而言,需要考虑的事情不是硬件能力不够,而且考虑能够充分将硬件利用起来。

内存、硬盘、CPU基本上可以认为是足够用的。对于服务端而言128G内存、4个intel至强核、2T 磁盘配置RAID10阵列卡可以认为是标准配置。

目前一套券商集中交易柜台订单笔数基本上在几万到几十万左右。所以在系统设计时,可以适当给系统增加一些容量限制,无限容量的系统设计时考虑的东西太多,反而影响了系统的设计。但是容量同样不能限制死,需要考虑未来的扩展

* 整套交易系统遵循以空间换时间的原则。
* 能在启动时初始化的数据就不在运行时动态申请
* 系统容量在启动时决定,可以通过配置动态扩展
* 能用代码、脚本搞定的事情坚决不用人来完成,能自动做的事情坚决不手动做
* 整套系统应该是一套异步事件驱动系统
* 整套系统必须拥有持久化功能,非实时数据、隔夜数据必须持久化而且还需要考虑定期归档。券商一般要求留存10年(印象中是,不确定)的业务数据。
* 系统需要考虑拥有重演机制。
* 系统应该是让客户可以自己定制的,也就是说系统只负责底层构建,上层的业务除了提供标准模板之外,允许客户自定义。例如风控指标、流程节点、权限管控

## 交易系统设计目标

### 系统功能

* 微秒级交易系统。纳秒级系统我个人认为除非考虑重写协议栈,由用户态来托管网络报文,将内部业务逻辑简化到极致,不采用异步事件驱动转而采用CPU100%轮询的方式进行数据的收发处理，否则比较难以达到。而且目前现货系统有集中交易柜台、交易所小站等开销情况下,纳秒级并无多大意义,除非为期货交易系统定制化。
* 对于操作行为应该具有一定的灵活性,特别是流程设置,如果能将流程设置变成和拖拽控件一样简单,那么对系统的使用来说是非常便利的
* 具有极大自动化的系统,以减少人为干预为目标。包括系统配置、定时重启、系统结构调整等等都应该采取标准化模板的方式。
* 为了保证数据安全,采用双重协议的方式进行数据隔离。也就是通过gateway进行协议转换。目的在于内外协议剥离,同时内部协议专注于效率,外部协议专注于兼容性、传输速度等。

### 业务功能

* 动态权限？一套交易系统里面由多个参与人：产品管理人、系统管理人、交易员、风控员等等。对于交易数据、持仓数据、盈亏数据等等权限是不一样的,需要能在一套标准模板的基础上动态设置。
* 审核体系 交易、风控等等环节均有大量的审核环节。要求权限控制、流程控制等功能
* 多账户同时交易支持拆单算法
* 支持定制化风控设置

## 交易系统设计结构

### 体系一(数据集中化):

恒生金证基本上采用了这个体系,以数据库为核心,将所有的数据集中到一个节点。

体系的优点是:数据集中,各个模块之间的交互快,业务拓展相对也比较方便。缺点很明显:数据集中依赖一个点,相互资源竞争强烈。

对这个体系我考虑了一个办法就是增加一个半持久化的数据模块。这个模块全部存储在内存里面,而且进程崩溃也不对导致数据丢失。数据的存储方式采用格式化的数据。通过行宽度对齐降低Fase Sharing效应,充分利用起系统的并发性。

半持久化的宿主我考虑是linux的共享内存。有点很明显,只要主进程创建好了共享内存,其他进程可以充分使用共享内存的数据,对于加锁的地方采用信号量控制。

在名策针对行情就是这么处理的,效率还是很高的。即使某一个进程崩溃了,只要再次附加共享内存就可以,不会有进程崩溃数据立刻丢失的问题。适合作为白天数据以及历史数据汇总的持久化载体。

这种模式的缺点是只能单服务器部署,多节点部署会比较麻烦,需要将不同的业务数据拆分到不同的服务器。当然对于拥有统一风控需求的金融交易系统来说,多节点平行运行本身可能就是一个伪命题。

对于统一网关,需要支持的功能是：

1. 网关只负责编解码、转发,无状态处理。
2. Session规则统一,多个网关之间都可以分别识别各自产生的session(便于分布式部署)



### 体系二(离散式):

这种模式的特点是各个模块相对比较独立,适合分散开发各自管理各自的业务逻辑和模块。部署起来相对灵活,可以多节点分机器部署。

这个模式的缺点在于,各自的数据没有一套有效的管控方式,很容易导致各个模块数据各自为政,任何的数据交互都必须通过网络协议进行,性能上稍微有一些影响当然这个对性能的影响不是重点。同时这种分散式的数据结构实际操作起来并不是很好驾驭。

目前市面上比较常用的高速缓存例如redis/ssdb等都是NOSQL类型的,存储基本上以key-value为主,对于订单、持仓、以及一些权限类数据并不是非常适合。



# 业务篇

### A股市场业务体系概要



### 权限体系

对于权限体系,现有产品比较常见的做法是针对菜单进行控制。目前恒生体系里面,初始的设计方案是只控制到了主菜单(包括一级菜单、二级菜单等)。随着信托基金业务的迅猛发展,需求人员发现主菜单控制体系已经不能满足现有业务需求。不同的客户针对菜单的权限松严有别。甚至有客户提出需要将独立的页面分开权限操作。

例如管理人-券商-私募产品体系里面,管理人由于有平仓权限(意味着管理人有充分账户、交易权限),私募公司认为,管理人不应当看到私募的持仓信息,因此要求管理人只能提供交易权限不能看到持仓页面。实际上现有的很多产品设计里面,为了方便交易员交易，持仓和交易基本上是同时展示的。

#### 角色划分:

投顾: 负责产品指令的实际发起,受到监管比较严格的监控

投资经理(投资总监): 负责实际审批投顾指令。目前产品监管比较严格,极少有能够绕开审批体系走自动审批的。

交易员: 负责交易指令执行。有一些自主管理产品里面也负责直接下单。监管地址基本上都是监管交易员的地址

风控员: 风控员负责产品风控的增、删、改。同时负责产品本身是否有风险的评估

查询员: 部分系统有这一类角色,主要目的是用于监控产品持仓、盈亏情况,这类角色没有任何交易权限,对账户只能只读。

管理员: 系统管理员或者超级管理员。基本上只负责系统日常运维、角色账户的设置等等

#### 角色功能:



### 交易系统

#### 算法拆单

##### TWAP:

时间加权平均算法。将一笔母单均匀拆分到各个时间批次。

例如母单为4000股,拆10次,那么每次400股均匀下达。主要注意的点:

1. 股票交易最小单位限制、最大单位限制
2. 定时器定时拆单
3. 定时撤单
4. 支持滑点、限价触发、条件触发
5. 考虑零股问题

##### POV:

成交量百分比算法。根据通过L2行情里面的逐笔成交(通过L1快照也可以)获取瞬时成交量,按瞬时成交量的百分比下单

1. 需要考虑扣除之前自己的成交量,否则pov的订单会越来越大
2. 零股问题

##### 冰山单:

和POV有些类似,区别点在于冰山单限定了每次下单的数量,每次成交之后,会补足下单的数量。没错发出去的都是小单,母单的总量市场是无法预知的。

#### 单笔委托

##### 单笔业务

目前市面上主要的业务有:普通股票、债券、回购、期指期货、单市场ETF、新股、配股、转股、投票、融资融券、期权等

交易体系属于整个系统最核心的部分。目前比较常见的做法是针对每一类型的委托增加一个业务方向的类型,针对每一个类型单独、顺序处理。一般处理的的流程主要是:

获取订单基础信息🡪权限检查🡪黑白名单、对敲检查🡪可用资金、持仓检查🡪资金、持仓预冻结🡪产生订单数据报盘

对于面向对象体系来说,处理单笔委托相对于关系型数据库来说具有优势。我考虑的主要有:

* 对于所有的交易可以建立元类,再元类的基础上,交易划分为两块: 买卖、借还。目前几乎所有的业务都可以用买卖、借还来进行划分。

对于买卖业务来说,基础的属性主要有委托要素(账号、价格、数量、业务类别)、订单资产、预冻金额数量。

对于借还业务来说,普遍会增加一个因素:利息以及可能带来的跨天业务。、

* 大量的基础数据可以通过指针的方式在元类预先注入,降低查找基础数据的开销。例如机构、资金账户信息、组合、代码等等。
* 对于交易业务而言,不是所有的业务都是需要在同一个线程中处理的,目前基本上账户可以作为交易的最小单位,对于不同的账户可以进行跨线程处理以提升系统并发能力。
* 对于运算量巨大的数据,特别是交易风控过程中的产生的数据开销,可以在订单下达时就汇总好,节省后续统一计算的开销,同时降低总的计算量
* 对于大部分数据,采用唯一序号的方式基本上都能做到O(1),当然以目前的运算能力unorder\_map也绰绰有余。

##### 订单管理

对于每一笔委托,都应该有统一的类、统一的线程来进行管理,采用从严控制的方式。

即:一笔订单一旦输入即认为有一笔订单,只有当次订单确定状态为废止状态时,才更变订单管理类。这样可以保证跨线程数据交互时,及时出现并发风险也是从严控制的。

订单类的职责:

1. 汇总订单数据包括: 订单产品、账号索引、状态索引等等。主要用于对敲等判断。

在关系型数据库里面,这里可以通过SQL索引来实现。对于内存数据库需要自行实现索引以提供个性化查询服务。例如B树索引。

1. 提供客户端查询服务。对于数据集中化处理可以直接查询集中数据
2. 订单管理可以考虑分账户式管理方式。即:

每个账号管理各自的数据,需要时统一进行汇总。



### 风控体系

#### 体系说明

目前风控体系的复杂之处在于:

* 不同的风控指标设置的数据不一样,对于资产类风控由于要考虑所有的资产,计算量极其庞大,导致资产类风控普遍速度并不快。对于庞大的数据计算每次都重复计算是非常损耗系统性能的
* 风控指标繁多,老架构里面基本上就是一个指标一个if else,导致风控的if else非常多。
* 无法支持客户定制,导致客户不断提自己需要的需求,导致风控愈加庞大

风控本质上是一批或复杂或简单的公式的集合。那么是否可以通过将公式化繁为简,对于原始数据,采用订单驱动的方式实时汇总；对于复杂公式采用逐个拆解的方式最终将复杂的风控计算转化为原始数据的简单拼接。

例如: 当前持仓占总资产百分比不超过XXX风控。拆解如下:



1. 拆解公式:

* 期初提供
* 客户根据固定格式进行配置

1. 拆解操作:

求和、求均值、求最大、求最小、取值

1. 原子数据:

代码中设定,后续根据需要逐个添加

1. 公式计算:

例如上面的例子就是:

(本次订单数量\*本次订单价格+(持仓数量1\*最新价格1)+(持仓数量2\*最新价格2) … ) / (当前余额+存款+场外资产+(持仓数量1\*最新价格1)+(持仓数量2\*最新价格2) …)

对于此类公式,直接将 本次订单数量、本次订单价格、持仓数量x最新价格x 替换为实际的数据之后就变成了一个数学计算公式。对于此类公式,一个堆栈计算器可以很简单的将这个做完,甚至复杂一点可以做一个语义解析器,能够直接帮客户识别客户编写的风控公式。即:风控最终是由客户来决定,系统仅仅是提供一系列的初始数据和操作。

#### 风控数据划分:

##### 分子/分母类

###### 分子

单只市值、集合市值(例如总仓位、按证券类别、按成分股、股票池、行业等)、

单只成本、集合成本、单只价格、持仓数量、净值、当日成交量

###### 分母

净资产、总资产、股本、市场成交量、委托总量、总市值、

##### 静态控制类

黑白名单、交易方向

#### 风控维度划分:

目前维度划分有:

迅投式: 全局风控🡪产品风控🡪账号风控

名策式: 市场风控🡪行业风控🡪个股风控

传统式: 公司层🡪部门层🡪项目层🡪账号层🡪资产组层

或者 按市值类、持仓类等风控类型划分

#### 风控审核:

风控由于其特殊性,基本上风控操作必定有审核环节。对于风控审核来说重要的点在于灵活可配置。例如按角色配置、按人员配置。同时提供可视化的界面。设置风控就像拖拽控件一样简单

### 账户体系

账户体系复杂点在于子账户体系非常难以管理,对于没有子账户体系的系统而言,账户体系相对比较简单。如下所示:



#### 清算体系

##### 分仓清算方式

分仓清算难点在于

* 需要实现自动化,往往券商清算非常晚(12点之后),对于清算人员而言急需一套能够自动化清算的系统
* 自动化的难点在于数据的不准确性,对于普通股票业务还好,目前各个券商接口不统一、实际数据甚至可能和业务文档出现巨大的偏差导致清算可能出异常
* 单账户清算相对而言可以比较简单(直接同步柜台数据)对于子账户而言,清算难度成指数级提升

清算的流程一般是:

接收流水🡪整理流水🡪交割🡪核算🡪对账

##### 当日数据清算方式

目前亦有部分系统采用当前的成交数据进行清算的。直接用白天的数据进行数据处理之后第二天直接和券商对账。此种方式比较简单,但是基本上很难保证准确。

特别是费用几乎做不到准确、还有分红配股等信息很难获取到。

# 技术篇

目前已经有想当多的开源通信库、开发框架等可供选择,对于此类主要方向在于选型而不在于从0开始编写一个框架,投入与产出并不成比例。

## 数据、协议、内存

证券交易里面大部分的数据都是定长的,所以目前很多协议直接采用结构体的方式存取以保证效率最高。其中的典型就是CTP柜台。对此我的考虑是,以结构体为蓝本改造一个数据元类,具体设想如下:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 格式 | 优点 | 缺点 | 备注 |
| 恒生协议1 | 以0x01分割的字符串,带header | 多行多列、字段灵活存取、解包直接解内存 | 报文比较大传输受影响、明文、无数据类型 |  |
| 恒生协议2 | 以0x00分割的字符串数据,带header | 多行多列、字段灵活存取、解包直接解内存、包含了数据类型 | 报文比较大传输受影响、明文 |  |
| 金证 | 以竖线分割的key-value串 | 多行多列、灵活存取 | 报文大、明文、无类型 |  |
| 金仕达 | 竖线分割的字符串 | 多行多列、解析方便 | 无列头、字段顺序严格、灵活性低 |  |
| 顶点 | 标准FIX |  |  |  |
| Protobuf | 自编解码 | 自动序列化、支持锯齿数据、带格式 | 解包慢(V3.0有改善) |  |

说明:

* 数据元类存在的目的主要在于方便管理,直接使用int double char数组也可以
* 人工定制的内存逻辑的道理是:

在我们的通信协议里面,其实绝大部分数据都是服务端在进行数据处理交互,客户端的报文本身并不多,每次协议的申请释放就带来了内存的开销,那么基于内存不值钱的考虑我是否可以直接建立一个初始缓存区域,每次申请好固定大小的内存链表,在**协议报文里面就进行内存申请,**对使用的人来说使用的人并不关心协议报文是怎么做的,所以协议报文里面的内存直接就用公共的内存复用块。



内存复用块结构如下,通过这种方式可以将内存开销降低到最小

* 每次将内存置为可用、不可用状态只要改链表头即可
* 内存可以无限复用
* 考虑到内存池实现太过复杂,这里面的内存地址设置为固定大小。例如 512B 1024B 4K 8K 32K 128K。再大的直接找操作系统申请
* 采用链表的方式可以动态扩展
* 申请内存、释放内存的接口和malloc free一模一样,因为free的时候送入char \*只要向前推取sizeof(struct MemCaChe)就是地址头,直接修改链表指向即可。



## 通信

### 概述

通信类功能我大体分为两大块:

* 一问一答
* 订阅发布

通信方式主要有:

* 跨线程通信
* 跨进程通信

对于通信模型,我比较倾向在服务端采用全有名方式通信。也就是说每一个连接都是有系统唯一名字的。每个连接的名字按照一个固定的方式进行组织。这样带来的好处是任何场景下的通信都可以通过对端的名字找到对方。同时也方便后续的路由层次扩展

对于订阅发布,目前需要考虑的是补线的功能。也就是CTP里面的公私有流的概念。

通信可以直接采用开源通信库。业务级别的通信库 例如zeromq、nanomsg、mqtt

#### 关于zeromq

Zeromq功能很强大,效率也很高。支持非常多的通信模型，同时也有相对成熟的社区。

http://zeromq.org/

### 追求极致的通信

目前有些开源项目也有为了追求超高的并发重写协议栈的,这类库的特点是最求速度的极致,但是并没有封装上层功能。例如dpdk以及腾讯基于dpdk开发fstack。这类库使用的成本相对还是很高的。需要重建业务模型。

## 异步事件框架

目前基于EventLoop的异步事件框架还是比较多的。Libuv/libevent/Nginx/Boost.Asio/Muduo/redis等等都可以直接用来搭建基础的业务框架。

目前大多数异步事件框架的响应程度还在毫秒级的阶段。但是对于业务使用来说已经基本上够了。

从考虑上看,直接将redis的事件驱动拿过来把Timer从链表修改为一个堆是比较靠谱的做法。简单而且里面东西很少。其他的框架由于考虑通用性都做了太多的逻辑处理,并没有必要。

## 结构

一个单元意味着一个线程,每个线程绑定一个独立连接。每个线程之间除了必要数据之外,应该尽可能的不用锁,将不同的账号数据放置在不同线程中,在实际的业务场景下是可行的。例如订单、拆单等都可以通过账号进行线程分离。不能避免锁,能降低一定程度的锁。同时,使用原子操作替代mutex同样也是一个可行的办法。例如zeromq里面几乎没有锁,对于临界区基本上都是通过原子操作完成的。



由框架管理了底层的一系列技术基础,业务库应该是一个单线程的库,不需要考虑线程间的数据处理。业务库本身是复杂的,所以需要框架提供更多的技术支撑。

