Simulated Exchange

项目报告文档

小组成员：

宋逸凡 **516030910411**

王见思 **516030910412**

潘子奕 **516030910239**

目录

一、需求分析

1.1 功能性需求

①Trader 登录/登出/注册：为 Trader 端提供登录/登出/注册功能。

②Trader 新建订单：Trader 选择指定 Broker 与指定 Future，选择订单类型、数量、价格等属性，可新建对应的 Market Order/Limit Order/Stop Order/Iceberg Order。

③Trader 获取行情与 OrderBook ：Trader 选择指定 Broker 与指定 Future，能够得到并实时更新行情与 OrderBook。

④Trader 查询个人历史订单：Trader 选择指定 Broker 与指定 Future，能够查询自己已经发出的订单。

⑤Trader 查询所有历史订单：Trader 选择指定 Broker 与指定 Future 与指定时间段，查询得到已成交的历史订单。

⑥Trader 取消订单：Trader 在个人历史订单界面选取订单点击取消，取消对应订单。

⑦Broker 管理 Futures：Broker 对所属的 Futures 进行查询/添加/删除等操作。

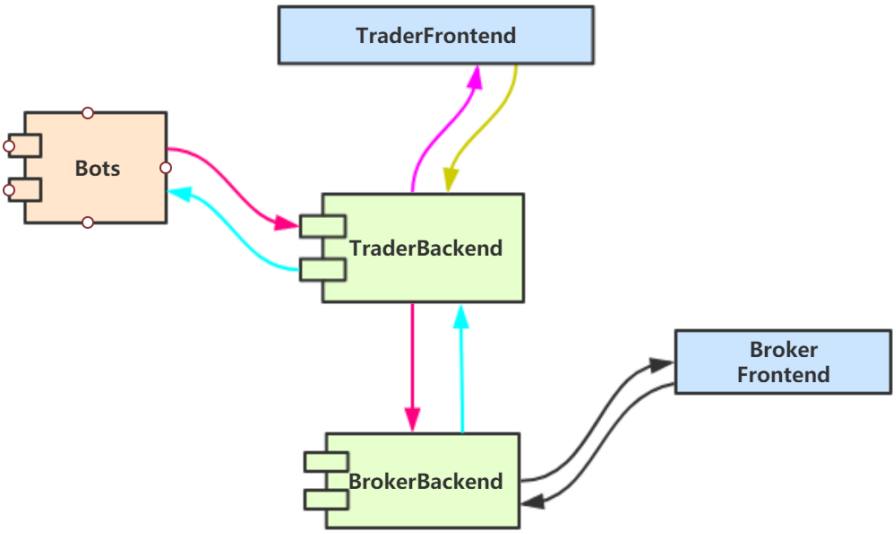
1.2 非功能性需求。

①可扩展性：TraderGateway 与 BrokerGateway 可以实现多备份与分布式架构。

②安全性：保证用户不会被冒充盗用，需要加密通讯信息。

③性能需求：保证满足基本的并发请求，行情订阅的延迟需要保证足够低。

④易用性：前端页面设计参考股票 app 与网页，所有功能使用步骤清晰明了。


TraderFrontend
Bots
TraderBackend
Broker
Frontend
BrokerBackend

系统通信简图

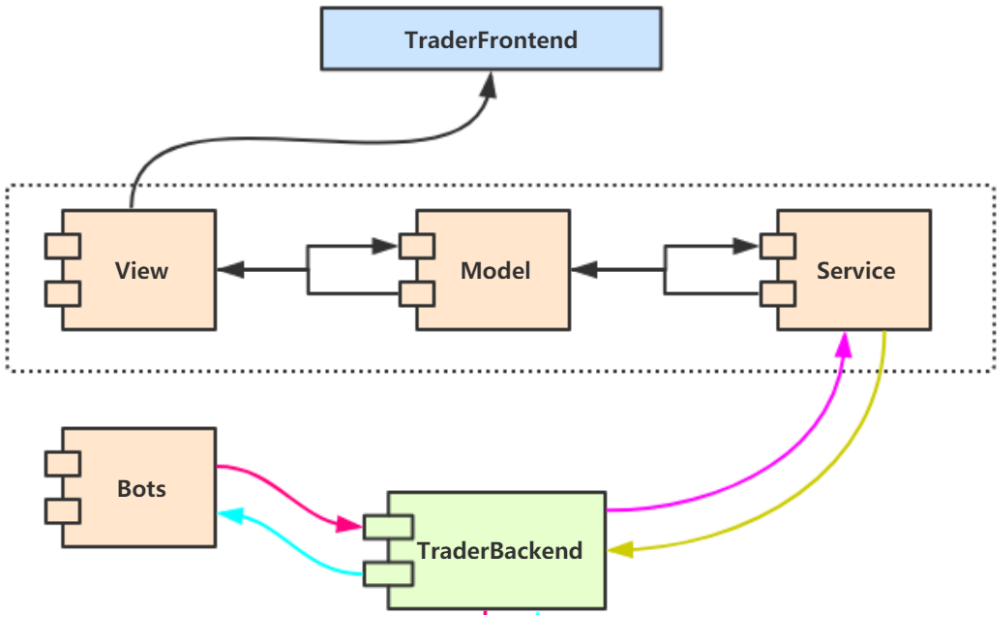
二、前端设计文档

2.1 整体架构

BrokerFrontend 与 TraderFrontend 均采用了 Antd-Pro 的框架，TraderFrontend 由于功能较多，采用的是旧版的整体完整框架，前者由于功能较为简单，采用的是新版空白框架。

BrokerFrontend 功能较为单一，方便起见将页面层、数据层与通信层合一。

TraderFrontend 则采取 Antd-Pro 框架中提倡的三层结构，即 Page 页面层，Model 数据层与 Service 服务层，这样架构较为清晰，各层间关系如图所示。



2.2 页面组成

TraderFrontend 主要页面包括：

交易中心页：查看指定 broker 中指定 future 的行情（通过 WebSocket 与 Trader-gateway 建立长连接，实时更新），与下单（包括 marketorder、limitorder、stoporder、icebergorder）。

——主要组件：Analysis/IntroduceRow/SalesCard/OfflineData

个人历史订单页：查看个人指定 future 的所有已建立的订单详情。

——主要组件：TableList

已完成订单页：查看指定日期内指定 future 的所有已成交订单的详情，只有涉及到本人的订单用户名详情才可以查看。

——主要组件：AllTableList

登陆页：提供登陆验证。

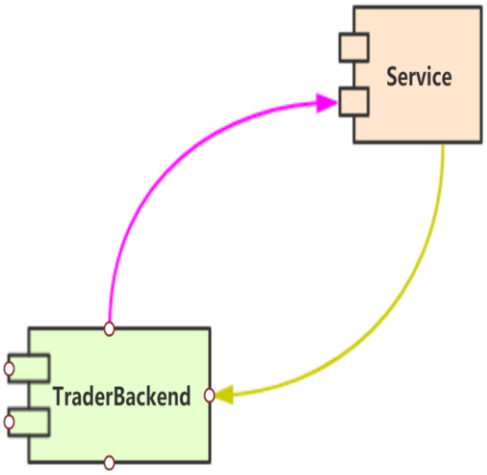
——主要组件：Login

BrokerFrontend 主要页面：

future 管理页：用于查看和管理制定 broker 的所有 futures

——主要组件：table-list 区块

2.3 通信



FrontendService TraderGateway

=>(once)

Username;Token;BrokerId;FutureId

<=(once)

History;OrderBook;MarketQuotatio

n

<=(cycle)

OrderBook;MarketQuotation

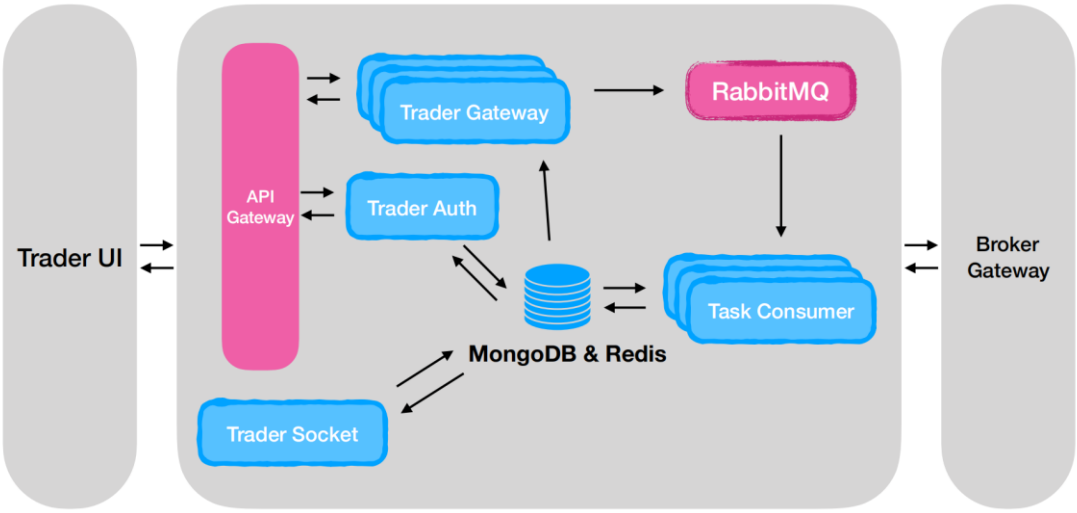
=>

Switch/Close

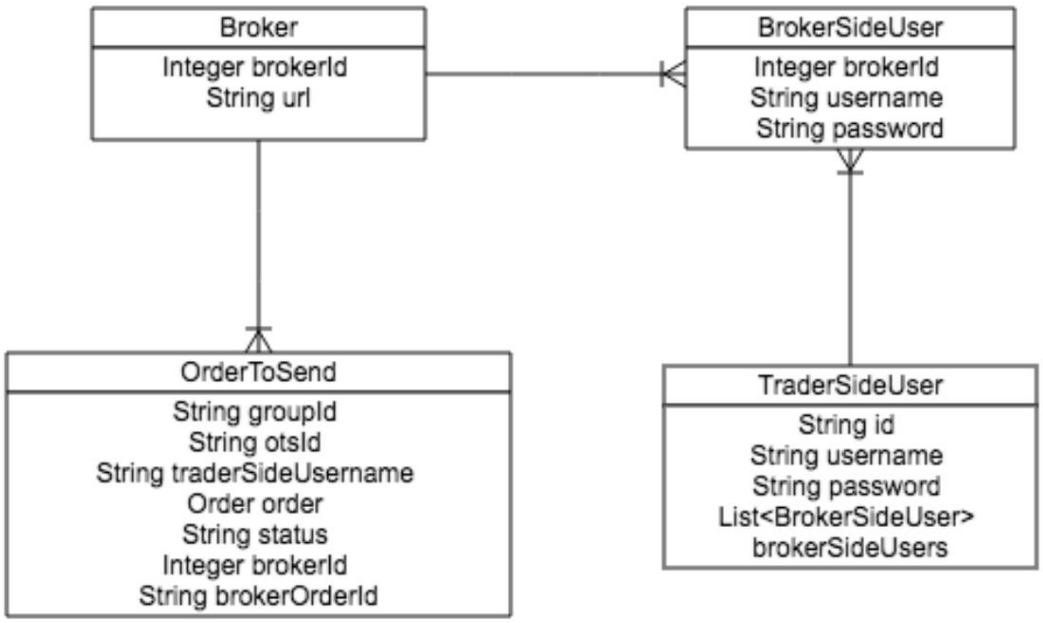
三、Trader-gateway 设计文档

3.1 整体架构：

1. Trader Gateway：主业务逻辑（包括下单、查看期货等功能）
2. Trader Auth：登录认证、用户信息编辑
3. Trader Socket：与 Trader UI、Broker Gateway 进行 socket 通信以及期货交易状态信息的存储
4. Trader Consumer：消费、取消 Iceberg 订单



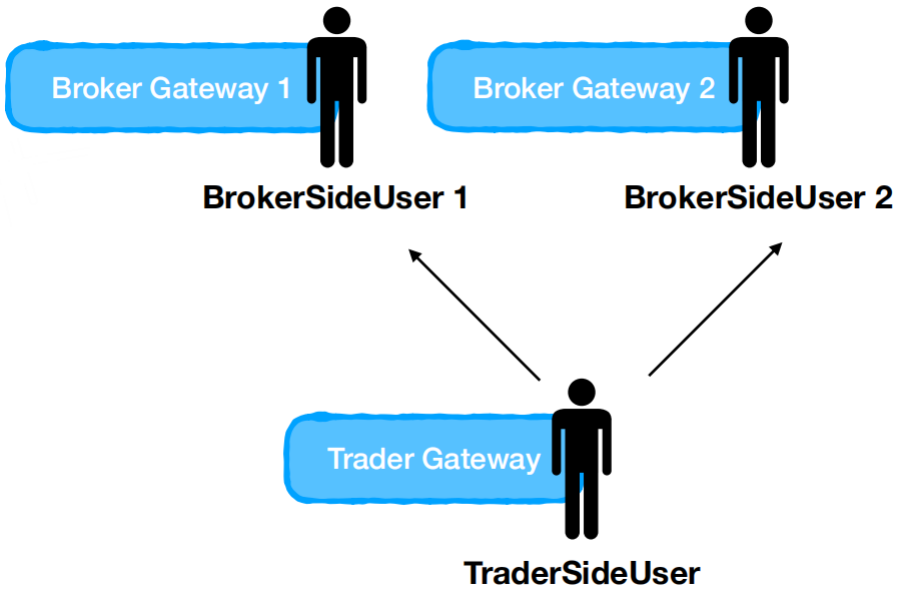
Trader-gateway ER 图



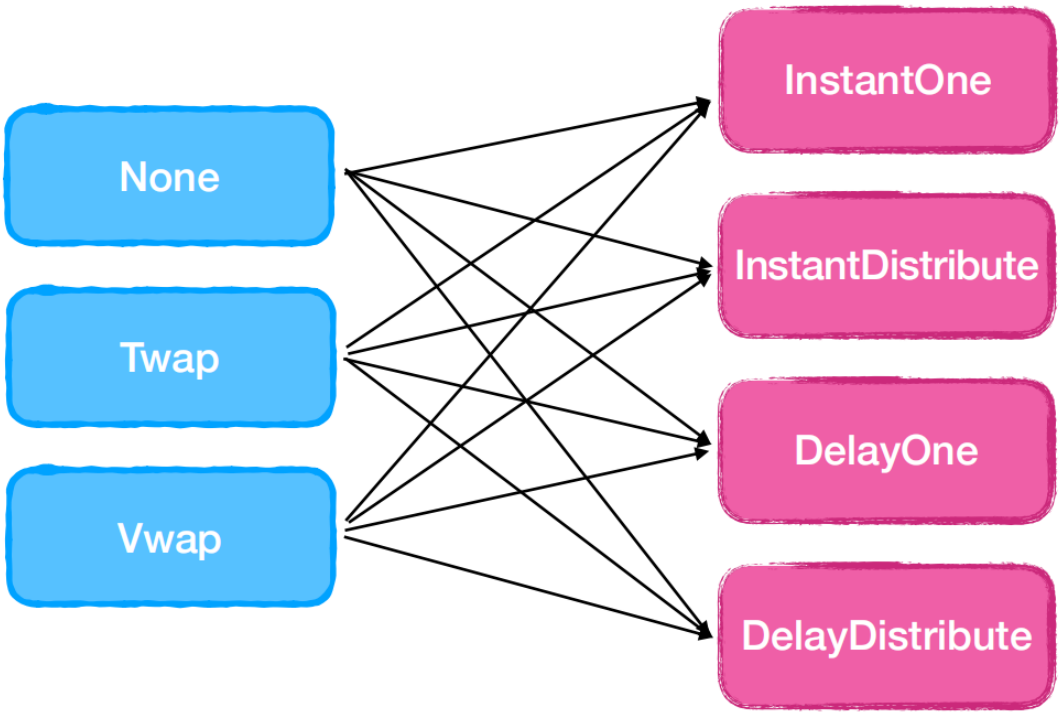
3.2 核心亮点

3.2.1 业务逻辑

1. 支持多个上游 Broker



2. 订单处理逻辑分为两步骤：处理与发送。我们一旦实现了某个新的处理或发送策略，都可以互相组合，形成新的交易策略



3. 支持 Iceberg 订单

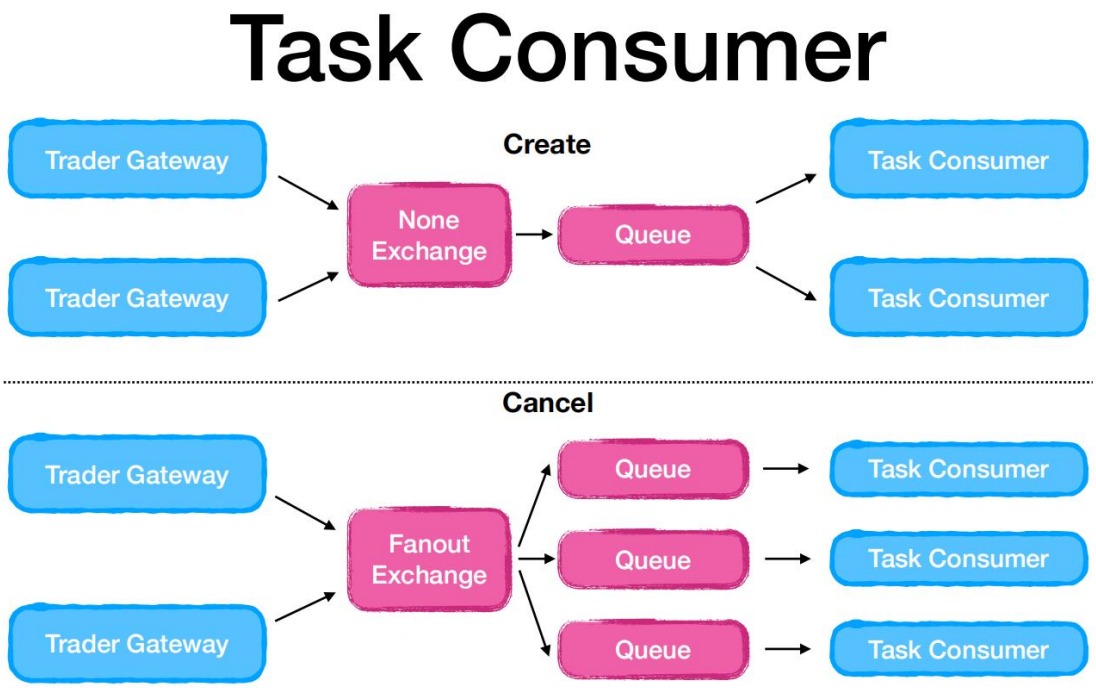
4. Iceberg 订单 Scheduled 或 Created 状态均可以被取消

3.2.2 架构设计

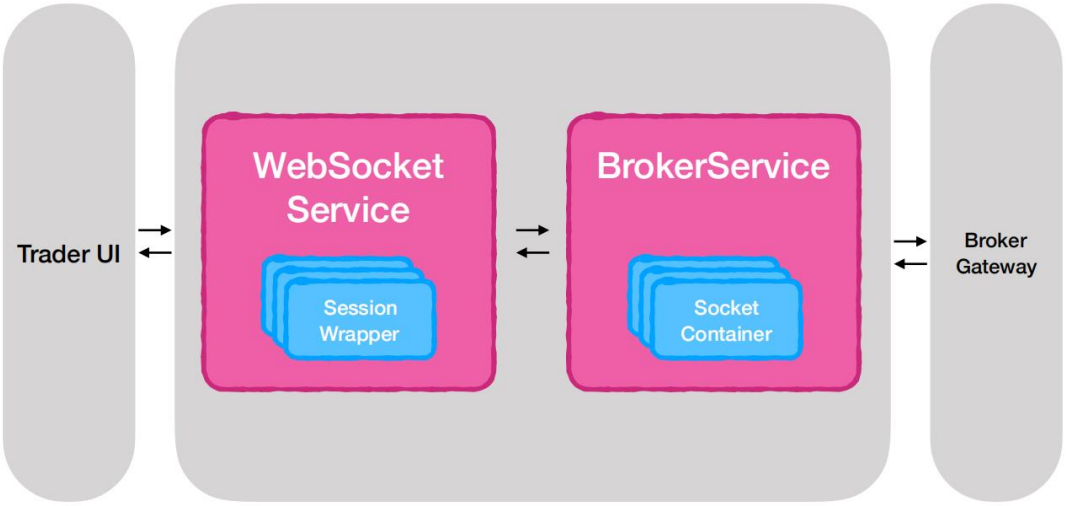
1. 服务拆分。将 socket 通信与登录拆分为单独服务、使用消息队列将 Iceberg 订单的消费者部分单独拆出。降低服务间的耦合性。

2. 高可用。服务拆分后，使得 Trader-gateway 成为无状态服务，支持高可用；Task-consumer 虽然是有状态服务，但有消息队列的存在，他的状态不会丢失，也支持高可用。

3. 使用消息队列后，根据我们实际业务逻辑，解决了重复消费与消息丢失的问题。

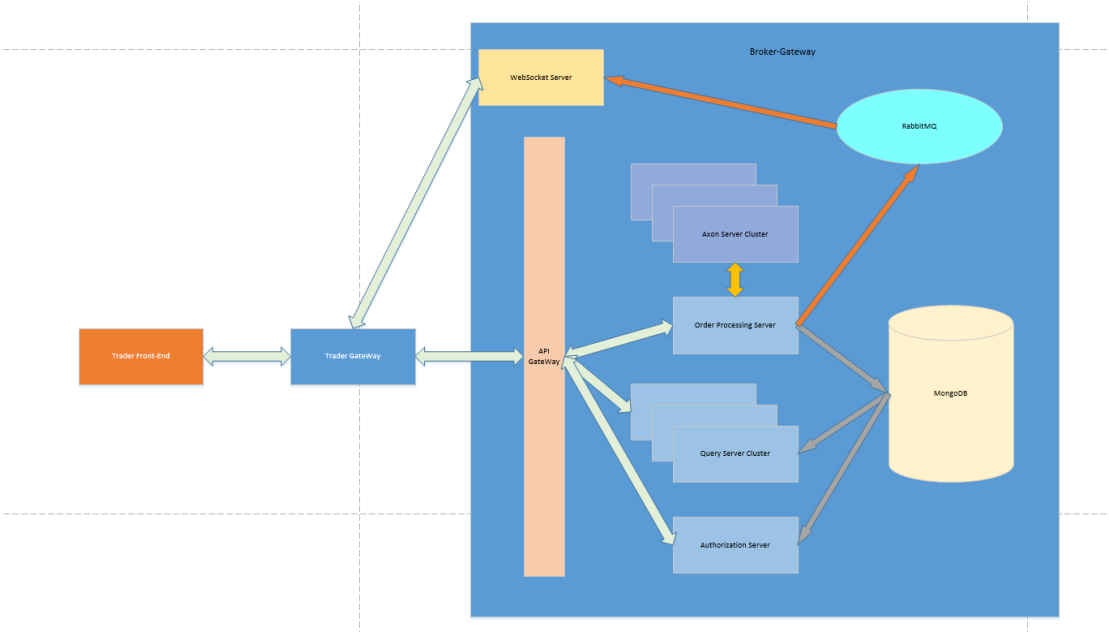


4. 一个 TraderUI 与一个 Trader-gateway 之间、一个 Trader-gateway 与一个 Broker-gateway 之间均只保持一个 Socket 连接进行通讯

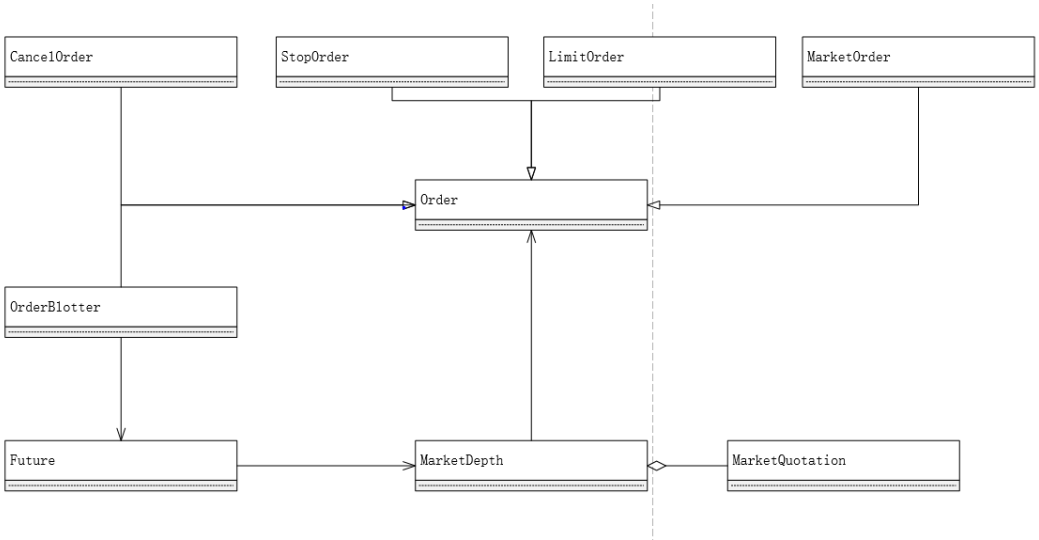


四、Broker-gateway 设计文档

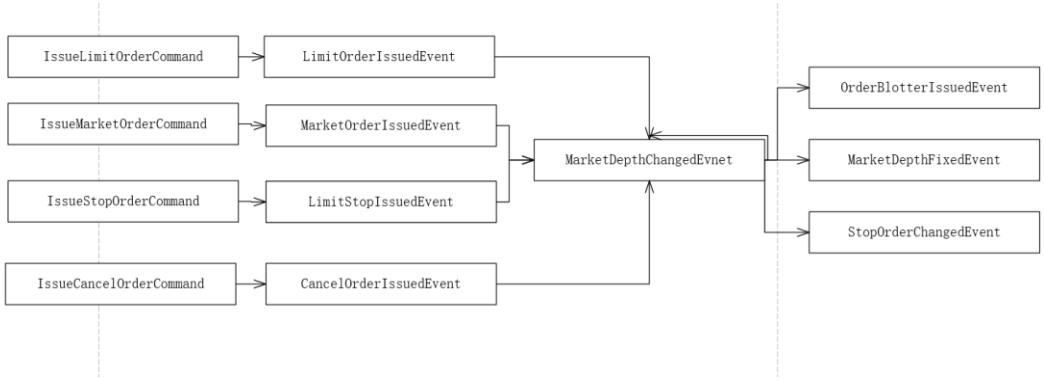
4.1 整体架构



Broker-gateway 类图：



Broker-gateway 事件驱动流程图：



4.2 核心亮点：

1. 低耦合：

1.1 使用了微服务架构，将各个独立功能点分离开来，可以独立部署，更好的隔离性和更细的部署粒度。

1.2 使用了 RabbitMQ 作为消息队列中间件，将服务之间消息的传递进行解耦。

1.3 使用了事件驱动的架构，通过 publish-subscribe 的设计模式进一步降低了各个组件之间的耦合，每个消费者只需要订阅相关事件即可。

2. 高性能：

2.1 通过 Axon 框架的消息机制实现了无锁的细粒度并发，允许订单的处理符合价格优先，时间其次的核心概念。

2.2 使用了纯内存订单处理，不需要将订单存到 Redis 或者 Mongo 这样的持久化数据库中，而是记录事件，从而减小了存储所需的耗时。

2.3 使用异步方法调用以及事件驱动，可以提高系统的响应时间，给用户带来更好的体验。

2.4 使用了二分搜索以及插入的订单处理算法，以及节点+链表的数据结构，可以将搜索以及处理时间复杂度从 O(n)降低到 O(logn)，从而获得更优秀的订单处理效率。

2.5 使用 CQRS 的设计模式，订单的处理和持久化数据查询的分离，可以提供更好的性能。

2.6 使用了 websocket 来对于部分信息采用主动推的方式，通过长连接节省了资源，并避免了不必要的轮询浪费

3. 高可用：

3.1 因为使用了微服务的架构，以及微服务无状态的设计，所有微服务都可以集群部署，从而保证了负载均衡带来的高性能同时，也带来了良好的容错、高可用性能，除非所有的集群都挂了，系统才会崩溃。

3.2 因为使用了事件驱动的架构，各个组件具有相当的自治性，哪怕别的组件挂了，当前组件也可以正常工作。

3.3 使用了事件溯源的设计概念，所有的数据都存在内存中，并针对事件进行持久化，当服务器崩溃时可以通过回放事件来进行恢复，同时采取了快照的策略，从快照进行回放可以减少回放时间。

3.4 使用了 docker-compose 以及 CICD 对于部署进行管理，允许一键部署。

五、Trade-Bots 设计文档

5.1 整体架构

本次项目中设计了四类自动下单的 bots，分别为：

①随机下单 bot：用于模拟市场的流动性，根据订阅的当前最新的行情在一定范围的价格内下单。

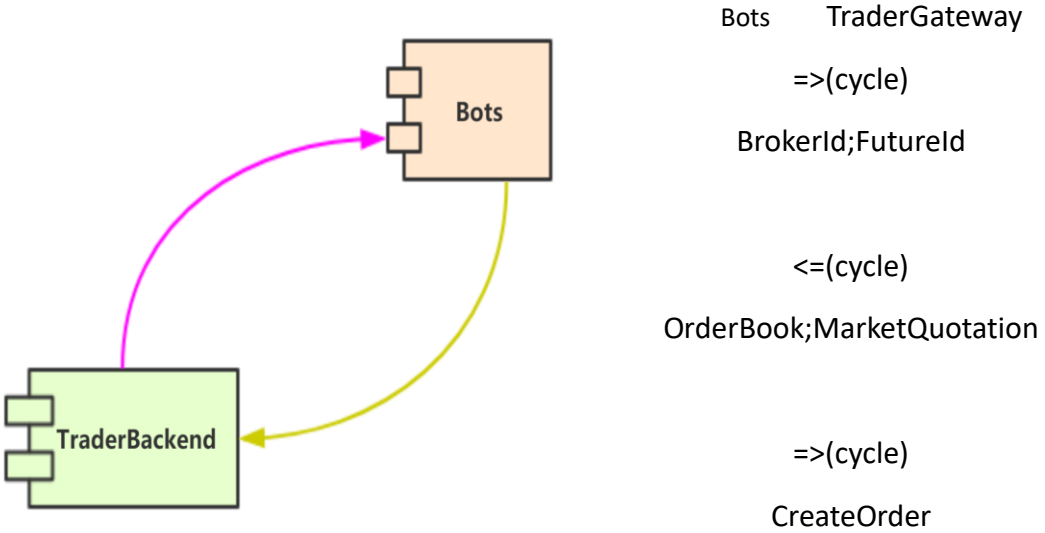
②趋势制造 bot：用于模拟市场的波动性，根据两个周期：长周期与短周期分别创造趋势，具体方法为根据订阅的当前最新的行情再带有周期更新的周期参数在一定范围的价格内下单。

③趋势跟随 bot：用于模拟采取一类策略的交易者，根据订阅的最近几期行情进行线性回归估计趋势，再根据最新的行情再带有趋势参数在一定范围的价格内下单。

④趋势逆反 bot：与③类似，仅为趋势参数的计算为与市场趋势反相关。

具体实现架构为定义一个 ACCOUNT 类，在登录并获取所属的 broker 对应的 futures 信息后，创建三个线程，分别用于更新行情、进行买卖方的下单。

5.2 通信



六、项目总结与收获

在前端开发过程中，我进一步加深了对于前端的层次化架构的理解，体会到了将数据层独立出来的好处：在 React 项目的构建过程中，时常需要在一个页面中使用多个组件，而不同组件之间的数据通讯也是很常见的，将数据层独立出来，再把修改/获取数据的函数封装为统一的接口可以使得组件间的数据传输更为简便，使得组件间的联动更加优雅等。不过我也发现，在项目体量不大的情况下，获取采取单层架构也是可取的方案，尤其是与后端的交互不多，数据较为简单的情况。此外，在与后端的通信中还增进了对于 WebSocket 的使用与特性的熟悉程度，例如如何判断连接状态以及根据连接状态执行对应逻辑等。

在 Bots 开发过程中，我对于真正市场中的角色有了新的认识，一开始我只是想写一个随机发单的 bot，但后来发现这样模拟出的曲线非常不真实，于是再进一步仔细思考，从而构造出了当前的四种 bot，效果较一开始大有进步。

在 trader-gateway 开发中，作为 broker-gateway 与 trader-ui 之间的中介，起初仅仅是快速搭建一个单体服务。但是在开发过程中逐渐意识到顶层架构设计与内部各个包的逻辑设计的重要性。其中对于订单处理部分的逻辑、以及访问 broker-gateway 的 Dao 层设计应用了一些设计模式使之更易维护，更具拓展性，使新增 Dao 层以及订单处理逻辑更加简单。其中大部分数据范围都在内存中做了 LRU 缓存。同时在开发过程中逐渐将 trader-gateway 中的组件从中分离，将状态组件分离，将天生异步的 Iceberg 订单服务使用消息队列实现，并且解决了消息队列使用中重复消费以及消息丢失的问题，对于消息幂等性有了更深入的理解。

在 broker-gateway 开发中，我学到了很多。一开始我设计的是一个只是简单的使用锁来控制并发，并为了持久化将所有的中间过程存在 Redis 中的一个单一系统，但是这样的一个设计使我的系统性能变得非常差，因为细粒度并发控制就意味着大量锁的阻塞，而将中间过程存在 Redis 就以为一个极大的带宽的出入。为此我研究了 LMAX 的架构设计，并引入了微服务设计、事件驱动设计、CQRS、Event Sourcing 和 Axon 框架，通过这些技术点，我成功得将原来高耦合的低性能的架构设计转变成了一个低耦合，分布式，高性能，内存处理的系统！这些技术

点为我打开了新世界的大门，让我看到了 DDD 的一种实现方向。

七、小组分工：

宋逸凡 516030910411：

TraderFrontend/BrokerFrontend/TradeBots 架构设计与实现

MarketQuotation 设计与实现

贡献度：1 / 3

王见思 516030910412：

Trader-gateway 架构设计与实现

贡献度：1 / 3

潘子奕 516030910239：

Broker-gateway 架构设计与实现

贡献度：1 / 3

日期：2019 / 06 / 19