Oedax 合约设计与使用说明

总体设计

合约主要分为:

- 拍卖子合约 —— Auction.sol
- 价格曲线合约 —— Curve.sol
- Oedax数据管理合约 —— Treasury.sol
- Oedax主合约 —— Oedax.sol
- 子合约生成合约 —— AuctionFactory.sol

主要功能:

- 拍卖子合约: Oedax拍卖功能，支持完整的拍卖的各项功能。
- 价格曲线合约:曲线计算，曲线参数存储，根据曲线参数计算。
- Oedax数据管理合约: 参与拍卖的用户账户管理，余额存放与提取。
- Oedax主合约: 拍卖合约的创建与注册，手续费相关设置，曲线设置。
- 子合约生成合约: 用于生成拍卖子合约，只能由Oedax主合约调用。

Oedax 合约功能实现

Oedax主合约

- 权限管理
  - 权限管理通过继承Ownable合约实现，具有onlyOwner修饰函数，owner权限转移等功能。
- 创建新的拍卖子合约
  - Oedax主合约中，拍卖子合约的创建通过调用`createAuction()`函数实现。用户输入初始参数调用`createAuction()`函数后，函数调用子合约生成合约创建新的拍卖子合约。合约间的struct数据传递采用`abi.encodedPacked()`函数进行序列化，序列化与反序列化功能在`DataHelper`合约中实现。
- 用户创建子合约查询
  - 用户创建子合约查询功能通过`getAuctions()`函数实现，具体实现逻辑是调用`treasury`合约中的接口，查询用户生成的合约地址，实现相应逻辑。
- 拍卖子合约的状态查询

    - 拍卖子合约的状态查询的功能由`getAuctionInfo()`函数实现，函数内部调用拍卖子合约的查询接口得到状态信息，反序列化后输出查询结果。
- 复制拍卖子合约
    - 对于已经拍卖结束的拍卖子合约，`cloneAuction`函数可以采用相同的参数创建一个新的拍卖子合约。其中，拍卖子合约采用的目标价格为已结束合约的最终撮合价格。
- 设置手续费参数
    - Owner通过调用`setFeeSettings()`函数设置手续费相关信息。
- 注册曲线合约（暂未使用）
    - `registerCurve()`与`unregisterCurve()`函数用于注册与注销外部的价格曲线合约。

Oedax数据管理合约

- 用户在合约中的账户余额管理
    - 用户在合约中的账户余额通过以下的映射数组进行记录
        - userTotalBalances - 纪录用户在合约中所有Token数量
        - userAvailableBalances - 纪录用户在合约中可用Token数量
        - userLockedBalances - 纪录用户在拍卖子合约中锁定的Token数量
    - 用户在合约中的Token信息通过一下映射数组进行记录
        - userTokenList - 用户所有记录过的Token地址列表
        - userTokens - 用户是否记录过Token
- 用户余额查询
    - 合约通过`getBalance()`函数查询用户的余额，函数返回用户对于特定地址的的Token总量，可用Token数量，以及被锁定的Token数量。
- 用户Token授权查询
    - 合约通过`getApproval()`函数查询用户在特定Token合约中的Token余额以及对Treasury合约的授权情况。
- 拍卖子合约的注册与记录
    - 合约通过`registerAuction()`函数注册新的合约，返回新注册合约的`ID`，该函数仅在Oedax主合约创建新合约的时候可以被调用。
    - 拍卖子合约的记录通过一下映射数组进行记录
        - auctionIDMap - 拍卖子合约的ID与合约地址的映射
        - auctionAddressMap - 拍卖子合约的合约地址与ID的映射
        - auctionCreatorMap - 用户地址与其生成合约的ID的映射
- 用户在Treasury合约中的Deposit操作
    - 用户在Treasury合约中的Deposit操作通过`deposit()`函数实现。函数中，Treasury合约向Token合约发起`TransferFrom`调用，在用户预授权的情况下实现账户充值的功能。充值成功后，用户对应的`userTotalBalances`与`userAvailableBalances`记录相应增加。
- 用户在Treasury合约中的Withdraw操作

    - 用户在Treasury合约中的Withdraw操作通过`withdraw()`函数实现。函数中，Treasury合约向Token合约发起`Transfer`调用，向用户转账。提款成功后，用户对应的`userTotalBalances`与`userAvailableBalances`记录相应减少。
- 拍卖子合约的Deposit操作交互
    - 拍卖子合约中用户在执行Deposit操作时，调用Treasury合约中的`auctionDeposit()`函数。拍卖子合约中用户的余额增加，Treasury合约中用户的`userAvailableBalances`减少，userLockedBalances增加，拍卖合约对应的`contractLockedBalance`增加。
- 拍卖子合约的Withdraw操作交互
    - 拍卖子合约中用户在执行Withdraw操作时，调用Treasury合约中的`auctionWithdraw()`函数。拍卖子合约中用户的余额减少，Treasury合约中用户的`userAvailableBalances`增加，userLockedBalances减少，拍卖合约对应的`contractLockedBalance`减少。
- 拍卖子合约的手续费交互
    - Treasury合约中的手续费交互操作有两种不同的情况，分别调用`sendFee()`函数与`sendFeeAll()`函数。
    - `sendFee()`函数
        - 手续费的分配为用户的单次操作，包括`walletFee`的收取以及`withdrawPenalty`的收取。调用该函数后，用户的`userLockedBalances`与`userTotalBalances`以及合约的`contractLockedBalances`相应减少，手续费收款人的`userAvailableBalances`与`userTotalBalances`相应增加。
        - 该函数只能通过Oedax主合约进行调用。
    - `sendFeeAll()`
        - 手续费的分配为合约结束时的整体操作，包括`protocolFee`与`creationFee`的分配，分配金额为合约所有参与量的对应比例的金额。调用该函数后，合约地址对应的`contractLockedBalances`减少，手续费收款人的`userAvailableBalances`与`userTotalBalances`相应增加。
- 合约紧急停止功能
    - Treasury合约具有紧急停止的功能，启用该功能后，该合约中的所有合约操作都将失效。用户可在合约紧急停止时，提取其存在Treasury合约中的所有余额。

子合约生成合约

- 根据参数生成拍卖子合约
    - 该功能由`createAuction()`函数实现，该函数只能由Oedax主合约调用，函数返回新生成的拍卖子合约的地址。

拍卖子合约

- 用户交易信息的记录

    - 用户在合约中的交易信息通过以下映射数组记录
        - users - 参与拍卖的用户地址的记录
        - userParticipated - 用户是否参与拍卖的记录
        - participations - 合约中交易信息的记录
        - participationIndex - 用户参与的交易信息编号的记录
- 拍卖状态的记录
    - 拍卖中交易Token数量的记录
        - askAmount - 用户存入TokenA数量的记录
        - bidAmount - 用户存入TokenB数量的记录
        - askQueue - 用户存入TokenA在等待序列中的记录
        - bidQueue - 用户存入TokenB在等待序列中的记录
    - 拍卖中交易状态的记录
        - status - 拍卖当前的状态，共五种状态
        - auctionState - 拍卖过程中的动态变量，有以下变量
            - askPrice - 当前卖出价格
            - bidPrice - 当前买入价格
            - actualPrice - 当前实际价格
            - totalAskAmount - 总共TokenA数量
            - totalBidAmount - 总共TokenB数量
            - estimatedTTLSeconds - 估计结束时间
            - queuedAskAmount - 卖单等待Token数量
            - queuedBidAmount - 买单等待Token数量
            - askDepositLimit - 最大TokenA存入数量
            - bidDepositLimit - 最大TokenB存入数量
            - askWithdrawalLimit - 最大TokenA提取数量
            - bidWithdrawalLimit - 最大TokenB提取数量
- 交易的Deposit功能
    - 用户通过调用`deposit()`函数将Token存入合约，参与拍卖。Deposit功能的实现分为以下几步：
        - 调用`updatePrice()`函数同步价格。
        - 调用Treasury合约中的`auctionDeposit()`函数锁定参与交易的Token数量。
        - 调用Treasury合约中的`sendFee()`函数，按比例收取walletFee。由于wallet地址很多，在用户Withdraw操作后不宜结算，所以不适合后置收取。
        - 调用`recordTaker()`函数记录Taker手续费相关数据。
        - 调用`updateAfterAction()`函数更新拍卖状态信息。
    - `updateAfterAction()`中的操作主要有以下几步：
        - 计算actualPrice到达价格曲线边界点需要的Token数量。
        - 超过到达边界需要的Token数量时，多出的Token会进入等待队列处理，具体的操作流程见**等待队列功能**介绍。
        - 更新拍卖状态信息。
- 交易的Withdraw功能

    - 用户通过调用`withdraw()`函数将Token取出合约，参与拍卖。Withdraw功能的实现分为以下几步：
        - 调用`updatePrice()`函数同步价格。
        - 计算用户可以提取的最大Token数量。
        - 调用Treasury合约中的`sendFee()`函数收取提款手续费`withdrawalPenalty`。
        - 调用Treasury合约中的`auctionWithdraw()`函数释放用户参与交易的Token数量。
        - 调用`updateAfterAction()`函数更新拍卖状态信息。
- 用户实时可以交易的 Token 数量范围计算
    - `getLimitsWithoutQueue()`函数用于计算不考虑**等待队列**时允许的最大交易数量范围，函数输入为当前的买入与卖出Token数量以及交易曲线在当前时间点的价格。
    - 合约通过`getLimits()`函数计算用户**不加入等待队列**情况下可以实时交易的Token数量，`getLimits()`函数的计算考虑了已有的**等待队列**。
    - `getLimits()`函数实现的算法如下
        - 首先调用`updatePrice()`函数更新当前的买入价格与卖出价格，时间同步点更新为当前区块时间。
        - `getLimits()`函数通过`getLimitsWithoutQueue()`函数辅助计算考虑**等待队列**时的交易Token数量范围。
- 等待队列功能
    - 等待队列用于处理因为价格限制而不能实时执行的交易，等待序列存储在数组`askQueue`与`bidQueue`中。
    - 用户参与拍卖时，deposit或者withdraw操作的Token数量超过实时`getLimitsWithoutQueue()`计算出的限制时，多出的部分会参与到等待序列的操作。以TokenA的操作为例，具体情况如下：
        - 情况1 - queuedAskAmount = 0，queuedBidAmount = 0
            - 对于deposit操作， `askQueue`中存入用户此次操作多出的Token数量。
            - 对于withdraw操作，由于函数限制不存在此种情况。
        - 情况2 - queuedAskAmount > 0，queuedBidAmount = 0
            - 对于deposit操作， `askQueue`中存入用户此次操作多出的Token数量。
            - 对于withdraw操作，用户取出操作中多出的Token，取出数量不超过 queuedAskAmount 。
        - 情况3 - queuedAskAmount = 0，queuedBidAmount > 0
            - 对于deposit操作， `bidQueue`中的Token根据当前的实际价格计算出对应TokenA的数量。若deposit的TokenA数量不超过`bidQueue`中对应的数量，则取出对应数量的TokenB，数量比例为当前实际价格的TokenA和TokenB同时加入拍卖中。若deposit的TokenA数量超过`bidQueue`中对应的数量，`bidQueue`中的Token清空，多出的TokenA加入到等待队列中。
            - 对于withdraw操作，由于函数限制不存在此种情况。
- 用户参与情况记录

    - 用户参与拍卖的交易信息记录在users、userParticipated、participations 、participationIndex映射数组中，已在上文介绍其对应功能。
    - `getUserParticipations()`与`getParticipations()`函数用于查询用户参与的交易。
- 手续费收取与分配
    - 拍卖过程中收取的手续费有以下几种:
        - creationFee - creator收取，在拍卖结束时分配
        - protocolFee - recepient收取，在拍卖结束时分配
        - walletFee - wallet或者recepient收取，用户存入时分配
        - takerFee - 所有用户根据权重分配，在拍卖结束时分配
        - withdrawalPenalty - recepient收取，用户提款时分配
    - withdrawalPenalty分配方法
        - 用户在拍卖过程中提款，按比例将手续费通过调用Treasury合约中的`sendFee()`函数从用户账户发送到recepient的账户中。
    - walletFee分配方法
        - 用户在Deposit操作时，按比例将手续费通过调用Treasury合约中的`sendFee()`函数从用户账户发送到wallet或者recepient的账户中。手续费结算后的Token数量作为实际参与拍卖的Token数量进行后续的操作与拍卖状态更新。
    - creationFee与protocolFee分配方法
        - 由于同一个拍卖中，手续费收取的比例一致，因此手续费的分配可以在拍卖结束时一起结算，调用`triggerSettle()`函数进行分配。分配方法是首先按比例计算出应该分配给creator与recepient的Token数量，然后通过Treasury合约中的`sendFeeAll()`函数进行分配。
        - 当拍卖`CLOSED`状态时，用户才可以调用`triggerSettle()`函数，函数实现内容如下:
            - 根据等待队列中的情况，返还等待队列中的Token。
            - 根据手续费比例，分配creationFee与protocolFee。
    - takerFee的分配方法
        - takerFee在用户最后提款时进行分配，用户所得为个人权重除以所有权重，然后乘以所有的takerFee对应的Token数量。具体的计算在`calcActualTokens()`函数中实现。
- 价格同步
    - 由于合约中拍卖状态的更新由交易触发，因此存在合约状态并非当前时刻的状态的情况。`updatePrice()`函数用于更新价格曲线到当前时刻，askPrice与bidPrice更新后，拍卖状态相关的其他变量更新（estimatedTTLSeconds, lastSynTime, 交易数量限制等）。状态更新后，函数判断当前拍卖价格（askPrice, bidPrice, actualPrice）是否满足拍卖状态跳转，如果满足则进行状态跳转。
- 剩余时间计算
    - 合约通过`getEstimatedTTL()`函数计算拍卖结束的时间，具体的算法是假设买入价格与卖出价格在未来的时间里都没有暂停，计算买入价格曲线与卖出曲线相交所需的时长，函数里采用二分法进行查找。

- 拍卖信息查询
  - 拍卖子合约中提供的查询接口有`getAuctionSettings()`, `getAuctionSettingsBytes()`, `getAuctionStateBytes()`, `getAuctionBytes()`, `getTokenInfoBytes()`, `getFeeSettingsBytes()`。接口返回序列化后的bytes类型的结果，信息接收者通过反序列化得到拍卖信息。
- 事件记录与主合约同步
  - 拍卖子合约中`triggerEvent()`函数用于拍卖过程中Deposit操作，Withdraw操作以及等待序列变化时的事件记录。
  - `emitEvent()`函数用于合约状态变化事件的记录，记录的同时通过调用Oedax主合约的`emitEvent()`接口实现事件记录的同步。

价格曲线合约

- 创建曲线
  - 合约通过`createCurve()`函数创建新的曲线，即将曲线的参数记录在`curveParams`参数数组中，函数返回生成曲线的ID。
- 复制曲线
  - 合约通过`cloneCurve()`函数对现有的曲线进行复制。曲线的复制操作中，更改曲线中T与P参数，不改变曲线的收敛形状（即S、M参数），函数返回生成曲线的ID。
- 计算买入价格曲线
  - 合约通过`calcAskPrice()`函数计算买入价格曲线。函数根据曲线的ID，计算ID对应买入价格曲线在时间点的价格。
- 计算卖出价格曲线
  - 合约通过`calcBidPrice()`函数计算卖出价格曲线。函数根据曲线的ID，计算ID对应买入价格曲线在时间点的价格。
- 计算买入价格曲线反函数
  - 合约通过`calcInvAskPrice()`函数计算买入价格曲线反函数。函数根据曲线的ID，计算ID对应买入价格曲线中价格对应的时间点。
- 计算卖出价格曲线反函数
  - 合约通过`calcInvBidPrice()`函数计算卖出价格曲线反函数。函数根据曲线的ID，计算ID对应卖出价格曲线中价格对应的时间点。

Oedax 合约使用说明

合约部署

1. 部署ERC20合约（至少两个）

2. 部署Curve合约
3. 部署Treasury合约
4. 部署AuctionFactor合约
5. 部署Oedax合约
6. 在Treasury合约与AuctionFactor合约中设置oedax地址
7. 在Treasury合约中进行token存入操作
    - 在ERC20合约中对Treasury合约进行approve授权
    - 在ERC20合约中对Treasury执行deposit函数，在合约中存入token
8. 根据拍卖参数，在Curve合约中调用createCurve函数创建新的曲线，记下曲线id
9. 在Oedax合约中调用createAuction函数生成新的拍卖子合约

拍卖合约操作

1. 在生成的合约中执行deposit与withdraw等函数参与拍卖
2. 拍卖合约随时间与价格的变化进入STARTED、OPEN、CONSTRAINED、CLOSED、SETTLED状态
3. 合约结束后，执行triggerSettle()释放等待序列中的Token，分配手续费
4. 用户执行settle()完成两种Token的交换操作

已部署合约(Ropsten Test Network)

1. TokenA地址： 0x4a6d31700990c06d8c41db2e717376bedea58658
2. TokenB地址： 0x66a195e166b1625b9bd62237264d24b8b1c24945
3. curve合约地址: 0x476fbdc16dc88ab41203003e9aedfc3e41c799fe
4. treasury合约地址： 0x8d6a87fbda124b167074248b4cc7cdffcbb81b6d
5. oedax主合约地址： 0x88b264a3b586eba8031931ec7d3c6e2c7be6a6fa
6. 子合约生成合约地址： 0x9360cf168cbc44e53560ea9b7bd690fd0b38677f
7. 拍卖子合约地址： 0xDc82b80bDDd8fA0C593E2D72BB48435DEA9Bfb32