# 新版OpenSea-Seaport协议



## Seaport概述

- 2022年6月11日，opensea部署了新版的交易市场合约---Seaport（部署交易：https://etherscan.io/tx/0x34a28a0111a238347fe30a6c91b00437f8438357427c5e759a2f7577afe65144 ）
- Seaport 的交易模型跟之前Opensea采用的 Wyvern Protocol 一样，依旧是**中央订单簿**的交易模型。都是**由链下的中心化的订单簿和链上的交易组成。**其中链下的订单簿负责存储用户的挂单信息，并对订单进行撮合。最终的成交和转移 NFT 是由 链上的Seaport合约 来负责的
- Seaport官方文档 :https://docs.opensea.io/v2.0/reference/seaport-overview
- Seaport 是一个市场合约，用于安全有效地创建和执行 ERC721 和 ERC1155 代币的订单。 每个订单包含任意数量的供应商愿意提供的物品（“报价，offer”）以及任意数量的必须连同其各自的接收者一起接收的物品（“对价，consideration”）。
- Seaport Protocol 最重要的目的就是：降低用户的 gas 消耗，降低购买流程的复杂度，提供更多的交易方式（物物交换、捆绑交易、拍卖等）

## Seaport产生的背景-- 旧版的缺点

- 随着生态的发展，之前OpenSea采用的 Wyvern Protocol 的各种问题开始暴露了出来：
  1. 多年前开发的，代码最后维护的时间都是4年前了。
  2. Wyvern协议本身不是专门为 Opensea 专门量身定做的，有大量的资源浪费。
  3. gas 消耗多，280000左右（单个）。
  4. 不支持直接使用代币购买 NFT，购买的时候需要转换，进一步提高了gas消耗和购买成本。
- 为了解决这些问题 Seaport Protocol 被开发出来

## Seaport 协议的 关键点

1. 支持批量购买/卖出，**可以捆绑不同的资产（如提供 ETH/ERC20/ERC721/ERC1155 资产）以换取 NFT（以物换物）**
2. 使用大量的内联汇编来降低 gas 消耗，目前测试来看是 134000左右（单个）
3. 交易特定的 NFT：**可以通过Merkle Tree设置 NFT 必须具备的特定“条件”。**
4. 支持部分交易、指定接单者等

## 重要合约地址

- Seaport 1.1合约地址：
  https://etherscan.io/address/0x00000000006c3852cbef3e08e8df289169ede581
- ConduitController：
  https://etherscan.io/address/0x00000000F9490004C11Cef243f5400493c00Ad63#code
- Conduit示例合约：
  https://etherscan.io/address/0x78b1beac4ebebee6231f82a69ff22c4296a2788f#code
- Zone示例合约：
  https://etherscan.io/address/0x004c00500000ad104d7dbd00e3ae0a5c00560c00#code

# 重要概念

首先，需要明确一些Seaport中的概念：

- 主网币 和ERC20 称作 货币（currency），ERC721、1155则称作 商品
- 对于erc721、1155，identifier为tokenId

- offer：挂单者； consideration：接单者作为token的转出方

```
enum Side {
    // 0: Items that can be spent
    OFFER,

    // 1: Items that must be received
    CONSIDERATION
}
```

- zone：订单的接单者，或者实现isValidOrder()方法、用于验证msg.sender是否为合法接单者的合约

# Seaport 源码分析

## 卖方offerer 批准一笔交易的几种方式

1. 卖方为交易的发起方，即msg.sender==offerer
2. 卖方通过签名的方式、授权交易体
3. 卖方调用交易市场合约中的`validate()`，在链上授权一笔交易？？

## ConduitController vs Conduit合约

- Conduit(渠道)：作为用户代币approve的目的地址，被Seaport市场合约调用、负责转移用户的代币。与conduitKey组成键值对保存在Seaport市场合约中。
- 默认情况下（即当conduitKey设置为零哈希时），offerer 将直接向 Seaport 授予 ERC20、ERC721 和 ERC1155 代币批准，以便它可以在执行期间执行订单指定的任何转移。相反，用户选择使用渠道后、将approve给提供的渠道密钥相对应的渠道合约，然后 Seaport 指示该Conduit转移相应的代币。
- ConduitController合约负责创建更新Conduit，维护键值对关系。Controller创建Conduit合约时，使用conduitKey作为salt，因此可以根据conduitKey和Conduit合约字节码、直接计算出某一个Conduit的地址。
- 另外，每个Conduit合约中会维护一个channel列表，标识可以调用该Conduit合约的用户（只有open channel列表中的用户才能调用合约中的executeXXX(...)，进行token transfer）。**channel即用户地址**

Conduit 总结： Conduit 就是提供了一个权限管理的功能，通过设置 conduitKey，来限制代币的转移。只允许注册在 ConduitController 管理的 Conduit 上的 channel 才有权限进行转移 token。这样无疑为创造者、收集者和平台提供了额外的能力。

## 用户挂单

- 挂单时，是在链下对以下结构体进行了签名：

```
OrderComponents(
        "address offerer,",
        "address zone,",
        "OfferItem[] offer,",
        "ConsiderationItem[] consideration,",
        "uint8 orderType,",
        "uint256 startTime,",
        "uint256 endTime,",
        "bytes32 zoneHash,",
        "uint256 salt,",
        "bytes32 conduitKey,",
        "uint256 counter", //同一个counter可用于多个订单
)
```

## 接单：一次交易匹配一个订单

### fulfillOrder & fulfillAdvancedOrder

- 函数定义：

```
function fulfillAdvancedOrder(
    AdvancedOrder calldata advancedOrder,
    CriteriaResolver[] calldata criteriaResolvers, // 条件解析器：Merkle树的方式指定tokenId
    bytes32 fulfillerConduitKey, // 接单者 msg.sender 要使用的conduitKey
    address recipient  //可以指定offerItem的接收者，默认是msg.sender
) external payable override returns (bool fulfilled) {
    // Validate and fulfill the order.
    fulfilled = _validateAndFulfillAdvancedOrder(
        advancedOrder,
        criteriaResolvers,
        fulfillerConduitKey,
        recipient == address(0) ? msg.sender : recipient
    );
}


function fulfillOrder(Order calldata order, bytes32 fulfillerConduitKey)
    external
    payable
    override
    returns (bool fulfilled)
{
    // Convert order to "advanced" order, then validate and fulfill it.
```

```
    fulfilled = _validateAndFulfillAdvancedOrder(
        _convertOrderToAdvanced(order), // convert过程，不支持部分接单
        new CriteriaResolver[](0), // 不支持条件解析器
        fulfillerConduitKey,
        msg.sender
    );
}
```

- 方法说明：
  1. 该订单可包含任意个数（0个或多个）的offer Item、 consideration Item
  2. consideration 中各个item的接收者在ConsiderationItem内部指定，不一定是转给offerer的
  3. offerItem 的接收者由交易入参recipient指定（fulfillOrder不支持该参数，默认为msg.sender）
- 两个方法的区别是：
  1. fulfillAdvancedOrder 更高级，支持订单部分执行、支持条件解析器，支持设置token接收者；
  2. fulfillOrder 执行普通订单，不支持订单部分执行、不支持条件解析器；普通订单会作为特殊的高级订单进行执行。
- 函数的业务逻辑：

```
1. 添加重入锁

2. _validateOrderAndUpdateStatus：根据参数，验证订单，更新状态并计算订单哈希值orderHash、需要执行订单的分子numerator和分母denominator

（1）时间校验：startTime <= block.time <= endTime

（2）分子与分母校验：

numerator <= denominator && denominator != 0;

若numerator == denominator，需要支持部分执行(Support Partial Fills)

（3）对价项目长度校验以及计算订单哈希值orderHash:

    - 订单长度校验：orderParameters.consideration.length >= orderParameters.totalOriginalConsiderationItems，即交易入参中consideration 数组实际长度要大于或等于 挂单者挂单时的元素长度（因为接单时可能会追加元素）

    - 计算订单哈希值orderHash

（4）校验高级订单的有效性：订单类型为 2 或 3 要求 zone 或 offerer 是 caller 或者 zone 批准。

（5）校验订单状态orderStauts = _orderStatus[orderHash]，保证订单没有被取消并且是可执行
```

的

    - 订单未取消，即：保证orderStauts.isCancelled == false

    - 订单可执行，即：若orderStatus.numerator != 0，保证orderStatus.numerator < orderStatus.denominator。

（6）校验订单签名，即若orderStatus.isValidated == false，则调用_verifySignature函数校验挂单者的签名

（7）计算本次交易“实际能够执行”的订单的分子fillNumerator和分母fillDenominator

（8）更新状态变量orderStatus

3. _applyCriteriaResolvers：根据交易入参criteriaResolvers数组，对各个指定的item应用条件解析器（校验Merkle树，并替换tokenId）， 校验其他未指定的item中都是确定的tokenId，确保提交的待执行订单是有效的。

4. _applyFractionsAndTransferEach：以2-(7)中指定的分数值执行每个offer item和consideration item的转账

5. _emitOrderFulfilledEvent：发出一个表明订单已完成的OrderFulfilled事件。

6. 删除重入锁

### fulfillBasicOrder

- 函数定义：

```
function fulfillBasicOrder(BasicOrderParameters calldata parameters)
    external  payable  override
    returns (bool fulfilled)
{
    // Validate and fulfill the basic order.
    fulfilled = _validateAndFulfillBasicOrder(parameters);
}
```

- fulfillBasicOrder 方法更像是为了兼容旧版的 Wyvern Protocol。
- 基本订单，只支持`ETH / native / ERC20 <=> ERC721 / ERC1155`之间的交易；不支持条件解析器；可以支持部分接单。支持的订单类型共24种、如下所示：

```
enum BasicOrderType {
    // 0: no partial fills, anyone can execute  表示 接单者用主网币ETH 换取 ERC721
    ETH_TO_ERC721_FULL_OPEN,
```

```
    // 1: partial fills supported, anyone can execute  支持部分接单，任何人都可以接单
    ETH_TO_ERC721_PARTIAL_OPEN,

    // 2: no partial fills, only offerer or zone can execute 只有offerer或者zone地址或者调用zone合约中的isValide...()验证成功的人才能接单
    ETH_TO_ERC721_FULL_RESTRICTED,

    // 3: partial fills supported, only offerer or zone can execute
    ETH_TO_ERC721_PARTIAL_RESTRICTED,

    // 4: no partial fills, anyone can execute
    ETH_TO_ERC1155_FULL_OPEN,

    // 5: partial fills supported, anyone can execute
    ETH_TO_ERC1155_PARTIAL_OPEN,

    // 6: no partial fills, only offerer or zone can execute
    ETH_TO_ERC1155_FULL_RESTRICTED,

    // 7: partial fills supported, only offerer or zone can execute
    ETH_TO_ERC1155_PARTIAL_RESTRICTED,

    // 8: no partial fills, anyone can execute
    ERC20_TO_ERC721_FULL_OPEN,

    // 9: partial fills supported, anyone can execute
    ERC20_TO_ERC721_PARTIAL_OPEN,

    // 10: no partial fills, only offerer or zone can execute
    ERC20_TO_ERC721_FULL_RESTRICTED,

    // 11: partial fills supported, only offerer or zone can execute
    ERC20_TO_ERC721_PARTIAL_RESTRICTED,

    // 12: no partial fills, anyone can execute
    ERC20_TO_ERC1155_FULL_OPEN,

    // 13: partial fills supported, anyone can execute
    ERC20_TO_ERC1155_PARTIAL_OPEN,

    // 14: no partial fills, only offerer or zone can execute
    ERC20_TO_ERC1155_FULL_RESTRICTED,
```

```
    // 15: partial fills supported, only offerer or zone can execute
    ERC20_TO_ERC1155_PARTIAL_RESTRICTED,

    // 16: no partial fills, anyone can execute
    ERC721_TO_ERC20_FULL_OPEN,

    // 17: partial fills supported, anyone can execute
    ERC721_TO_ERC20_PARTIAL_OPEN,

    // 18: no partial fills, only offerer or zone can execute
    ERC721_TO_ERC20_FULL_RESTRICTED,

    // 19: partial fills supported, only offerer or zone can execute
    ERC721_TO_ERC20_PARTIAL_RESTRICTED,

    // 20: no partial fills, anyone can execute
    ERC1155_TO_ERC20_FULL_OPEN,

    // 21: partial fills supported, anyone can execute
    ERC1155_TO_ERC20_PARTIAL_OPEN,

    // 22: no partial fills, only offerer or zone can execute
    ERC1155_TO_ERC20_FULL_RESTRICTED,

    // 23: partial fills supported, only offerer or zone can execute
    ERC1155_TO_ERC20_PARTIAL_RESTRICTED
}
```

- 通过 fulfillBasicOrder 执行的订单要满足的条件：
  1. 该订单只包含一个 offer 项目，并且**至少包含一个** consideration 项目（多余的部分保存在 additionalRecipients[] 中）
  2. 该订单只包含一个 ERC721 或 ERC1155 项目，并且该项目不是基于标准的（Criteria-based，直接指定某一个tokenId、而不能使用merkle tree指定多个中的任一个）。
  3. 该订单的 offerer 会收到 consideration 中第一个项目
  4. 用作货币（currency）的 token（ETH/ERC20属于货币类型，ERC721、1155属于商品） 必须是同一种。也就是说要么是原生代币作为支付货币 要么是 ERC20 的 token 作为支付货币，**不能混合支付。**
  5. offerToken不能是原生代币。
  6. 每个offerItem或considerationItem结构体的 startAmount 必须与该项目的 endAmount 一致（即项目不能有升/降金额）----**不支持拍卖**
  7. 所有 "被忽略 "的offerItem或considerationItem结构体中的字段（即 主网币ETH时的 token 和 identifierOrCriteria 以及 ERC20 对应的item的 identifierOrCriteria）被设置为空地址或零。

8. 原生货币项目上的token需要设置为空地址，货币上的标识符需要为 0，ERC721项目上的数量要为 1。
9. 如果订单有多个对价项目，并且除第一个对价项目外的所有对价项目与被提供的项目类型相同，则被提供的项目金额不低于除第一个对价项目金额外的所有对价项目金额之和。？？？？

- 整体业务逻辑：

1. 从 BasicOrderType 字段中提取订单类型orderType（是否支持部分接单、是否限制接单者）和基本订单路由BasicOrderRouteType（标识挂单者和接单者之间转移哪种类型token），并且对其进行校验

2. 准备执行基本订单

（1）添加重入锁

（2）校验时间正确

（3）检验参数正确

（4）计算并校验订单的哈希值

（5）更新订单状态

3. 若使用了conduit，则根据订单路由选择使用offerer或fulfiller的conduit

4. 根据订单路由，执行原生代币以及ERC721代币的转账，完成订单

5. 删除重入锁

## 接单：一次交易匹配多个订单

### matchOrders & matchAdvancedOrders 撮合多个订单

- 对一组订单（大于等于2个）进行匹配，即订单和订单之间的撮合。示例如下：

目前有三个订单：
第 1 个订单： offer *---tokenA ，consideration---tokenC、tokenD*

第 2 个订单： offer *---tokenD、A ，consideration---tokenB*

第 3 个订单： offer *---tokenB、C ，consideration---tokenA*

- 如果这三个订单总的offer(token类型、数量) >= 总的consideration，那么这三个订单是可以撮合在一起的,即可以调用matchXXX.... (这里用的是>=是因为，比如用户A卖出一个NFT，标价为5ETH--- *offer:NFT,consideration:5ETH ；用户B想要买入该NFT，提供12ETH---*

*offer:12ETH,consideration:NFT。*
两个订单汇总后，offer：NFT+12ETH > consideration：5ETH+NFT===》可以撮合。因此只需要保证consideration侧能够match即可，offer侧有剩余也ok，，相当于帮用户A省了钱。实际链上会将用户A的订单 份额设置为1/1)

- 具体如何撮合由交易入参fulfillments指定，fulfillments 是 具体 每个订单 offer 中的项目与另外一个订单中具体哪一个项目进行匹配的信息：
    1. fulfillments[0]表示的内容是，将第1个订单的offer中的第1个item 和第3个订单中的consideration的第1个item匹配；
    2. fulfillments[1]表示的内容是，将第2个订单的offer中的第1个item 和第1个订单中的consideration的第2个item匹配；将第2个订单的offer中的第2个item 和第3个订单中的consideration的第1个item匹配
    3. fulfillments[2]表示的内容是，将第3个订单的offer中的第1个item 和第2个订单中的consideration的第1个item匹配；将第3个订单的offer中的第2个item 和第1个订单中的consideration的第1个item匹配

- 以这种方式fulfill的订单没有一个明确的接单者，msg.sender只是撮合者，并不是接单者！！！因此交易成功的事件中 recipient 为空。 recipient。需要根据多个订单综合考虑。
- 遇到校验失效的订单，会直接revert整笔交易。匹配订单时，所有订单的总offer应该等于所有订单的总consideration，否则匹配失败、revert。

```
function matchAdvancedOrders(
    AdvancedOrder[] memory advancedOrders,
    CriteriaResolver[] calldata criteriaResolvers,
    Fulfillment[] calldata fulfillments
) external payable override returns (Execution[] memory executions) {

    return
        _matchAdvancedOrders(
            advancedOrders,
            criteriaResolvers,
            fulfillments
        );
}


function matchOrders(
    Order[] calldata orders,
    Fulfillment[] calldata fulfillments
) external payable override returns (Execution[] memory executions) {
    // 作为matchAdvancedOrders的一种特殊情况，转为advance订单后再执行
    return
        _matchAdvancedOrders(
```

```
            _convertOrdersToAdvanced(orders), //将Order转为AdvancedOrder类型，不支持订单部分执行
            new CriteriaResolver[](0), // 不支持条件解析器
            fulfillments
        );
}
```

- matchOrders：对普通订单中的报价和对价项目按参数fulfillments 提供的报价组件分配给对价组件的条件求进行匹配然后执行，**不支持订单部分执行，不支持条件解析器**；普通订单作为特殊的高级订单进行处理
- matchAdvancedOrders：对高级订单中的报价和对价项目按参数fulfillments 提供的报价组件分配给对价组件的要求进行匹配然后执行
- 实现逻辑：

```
1. _validateOrdersAndPrepareToFulfill：

2. _fulfillAdvancedOrders：
```

- 函数的业务逻辑：

```
1. _validateOrdersAndPrepareToFulfill：校验订单（若有无效订单则回滚）、更新其状态、通过先前填充的分数减少金额、应用条件解析器并触发 OrderFulfilled 事件。若有无效订单，则回滚整个交易。具体步骤如下：
    1). 添加重入锁

    2). for循环遍历每个advancedOrder，依次执行：
    - 调用_validateOrderAndUpdateStatus：根据参数，验证订单，更新状态并计算订单哈希值orderHash、需要执行订单的分子numerator和分母denominator。
    - 根据1)中的百分比，计算出本次交易中每个offerItem、considerationItem的实际transfer数量--将这些数量保存在advancedOrder.parameters.offer或consideration.startAmount字段中（此处还不进行真正的transfer）

    3). _applyCriteriaResolvers：根据交易入参criteriaResolvers数组，对各个指定的item应用条件解析器（校验Merkle树，并替换tokenId）， 校验其他未指定的item中都是确定的tokenId，确保提交的待执行订单是有效的。

    4). for循环遍历每个advancedOrder，依次_emitOrderFulfilledEvent：发出一个表明订单已完成的OrderFulfilled事件。

    5). 删除重入锁
```

2. _fulfillAdvancedOrders：在验证、计算item实际transfer数量和应用条件解析器之后，执行高级订单：

1). 为每一个订单的报价和对价项目的执行分配一个Execution结构，构造为一个Execution结构数组executions

2). 循环遍历交易入参fulfillments ，将executions 中的每一个元素对应到fulfillments的每一个元素，若报价人者（offerer）和接受者（recipient）是相同的，则跳过。

3). _performFinalChecksAndExecuteOrders：对高级订单以及executions进行最后的校验，然后执行订单，完成执行转账，删除重入锁。

### fulfillAvailableOrders & fulfillAvailableAdvancedOrders 接单

- 这两个方法用来批量成交订单，一次性购买多个订单。类似于 gem 这类聚合器起到的作用。

#### fulfillAvailableOrders

```
    function fulfillAvailableOrders(
        Order[] calldata orders,
        FulfillmentComponent[][] calldata offerFulfillments,
        FulfillmentComponent[][] calldata considerationFulfillments,
        bytes32 fulfillerConduitKey,
        uint256 maximumFulfilled
    )
        external
        payable
        override
        returns (bool[] memory availableOrders, Execution[] memory executions)
```

- 如上所示，fulfillAvailableOrders包含两个数组类型的返回值，availableOrders标识的是参数orders中各个订单是否交易成功；executions中包含的是本笔交易中所有的ETH、ERC20、721、1155的转移情况
- 不能指定recipient!

## solidity知识点强调

- solidity中，memory 引用类型 赋值给另一个memory的引用时，是不会创建拷贝的。即memory之间通过引用传递完成。

```
function testMemory(uint256[] memory aa) public pure returns(uint256[] memory){
    uint256[] memory bb=aa; //bb和aa 指向的内存相同
    bb[1]=123; //对bb的修改和对aa是相同的
    return aa; //此时 aa[1]=123
```

```
}
```