从0到1部署Reactive合约

作为一名刚刚接触Web3不久，参加实习计划的小白，刚看到任务的时候是一头雾水，Reactive合约是什么，有什么用，如何编写或部署，完全摸不着头脑。通过两天课余时间的学习了解，也许可以将一点小小的经验分享给大家，帮助大家更好理解和实践。

Part1. 先来了解些背景知识

首先解决第一个问题：Reactive合约是什么
简单来说，Reactive能够做到两件事：监督某个合约的event是否发生，触发提前指定的callback合约的执行，而且，这两个合约可以是部署在不同链上的

那我们为什么需要Reactive合约呢?
大家应该还记得，常规理解中，合约只能由用户调用后才能生效，智能合约无法在没有外部指令的情况下发起操作，要想实现这种无需手工就能调用合约的效果，会不可避免地引入中心化的控制手段，而Reactive合约的出现可以很好地解决这一限制，它能够从多条链获取信息，去中心化地跨平台触发合约，这在需要实时反应地DeFi等领域有着十分广阔地应用前景

OK，来到最重要的部分，如何部署Reactive合约，真正实现跨平台地自动调用合约
再真正操作前，需要简单再补充几个概念

1. 前面提到，Reactive可以监控event，那event究竟是什么

简单来说，这就是你在合约里自己定义的，要发布在区块链交易日志上的信息
例如 event PriceUpdated(string symbol, uint256 newPrice);，就是定义了一个叫PriceUpdated的事件，会展示代币的symbol和newPrice，在代码中使用emit PriceUpdated("ETH", newEthPrice);将最新价格更新到交易日志中

一个典型的log类似这样

```
{
  "address": "0x...ORIGIN_ADDR...",
  "topics": [
    "0x8cabf3...a4cb",           // topics[0] = topic_0：事件类型
    "0x0000...<32字节>",         // topics[1] 可选：indexed 参数1
```

```
    "0x0000...<32字节>"          // topics[2] 可选：indexed 参数2
  ],
  "data": "0x....",              // 非 indexed 参数的 ABI 编码
  "blockNumber": "0x...",
  "transactionHash": "0x...",
  "logIndex": "0x..."
}
```

最关键的部分是topics和data，topics最多可以有四个，topics[0]永远是事件的签名哈希，[1-3]则是对应的indexed的参数，剩下的则放进data里，举个例子

event Received(address indexed from, address indexed to, uint256 amount);

此时，event中topics[0]=keccak256("Received(address,address,uint256)")

topics[1]=from（被编码为32bytes，左侧补0，右侧20bytes是地址）

topics[1]=to

amount因为没有indexed，直接被放进data

虽然一般区块链浏览器中返回的JSON格式的log，但真正上链的只有address，topics和data

2. Reactive合约部署在哪里?

Reactive合约并非部署在以太坊或其他链上，**每个Reactive合约包括两个实例（instance），一个在Reactive Network上，一个在ReactVM上**。这种并行的逻辑一方面可以提高效率和扩展性，另一方面很好的保护了安全性

①Reactive Network和EVM区块的运行很相似，有两处最大的不同，

一是允许跨链订阅其他链上的event

二是每个部署的地址都有一个对应的ReactVM

②React VM是一个 受限的虚拟机，与外部网络隔离，只能和它对应的Reactive Network地址的合约交互

因此，一个常规的Reactive合约的工作流程是：

- 被监测的外部合约发出特定事件（在订阅时会设置一个过滤器）
- Reactive Network收到事件，传递给对应的React VM
- React VM根据收到的事件，向Reactive Network发起请求

- Reactive Network收到请求，发起调用目标合约的交易

Part2. 先来一个最简单的小例子

终于，到了最关键的代码环节，这部分我们执行两个小实例，跟着 来具体操作一遍

先看源合约（也就是被监控的合约），逻辑非常简单，**当收到一笔钱后，会将钱还给发送者，并发出Received事件**

```
contract BasicDemoL1Contract {
    event Received(
        address indexed origin,
        address indexed sender,
        uint256 indexed value
    );

    receive() external payable {
        emit Received(
            tx.origin,
            msg.sender,
            msg.value
        );
        payable(tx.origin).transfer(msg.value);
    }
}
```

接下来看看目标合约，也就是将被Reactive合约调用的合约，同样非常简单，关键在于**callback函数**，要执行的逻辑也非常简单，就是发出一个CallbackReceived事件，但里面还有一些细节，比如authorizedSenderOnly是什么，rvmIdOnly(sender)又是什么，我们看完Reactive合约再回来解释

```
// SPDX-License-Identifier: GPL-2.0-or-later

pragma solidity >=0.8.0;
```

```
import '../../../lib/reactive-lib/src/abstract-base/AbstractCallback.sol';

contract BasicDemoL1Callback is AbstractCallback {
    event CallbackReceived(
        address indexed origin,
        address indexed sender,
        address indexed reactive_sender
    );

    constructor(address _callback_sender) AbstractCallback(_callback_send
er) payable {}

    function callback(address sender)
        external
        authorizedSenderOnly
        rvmIdOnly(sender)
    {
        emit CallbackReceived(
            tx.origin,
            msg.sender,
            sender
        );
    }
}
```

最后是Reactive合约，整个系统的核心

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity >=0.8.0;

import '../../../lib/reactive-lib/src/interfaces/IReactive.sol';
import '../../../lib/reactive-lib/src/abstract-base/AbstractReactive.sol';
import '../../../lib/reactive-lib/src/interfaces/ISystemContract.sol';

contract BasicDemoReactiveContract is IReactive, AbstractReactive {
```

```
    uint256 public originChainId;
    uint256 public destinationChainId;
    uint64 private constant GAS_LIMIT = 1000000;

    address private callback;

    constructor(
        address _service,
        uint256 _originChainId,
        uint256 _destinationChainId,
        address _contract,
        uint256 _topic_0,
        address _callback
    ) payable {
        service = ISystemContract(payable(_service));

        originChainId = _originChainId;
        destinationChainId = _destinationChainId;
        callback = _callback;

        if (!vm) {
            service.subscribe(
                originChainId,
                _contract,
                _topic_0,
                REACTIVE_IGNORE,
                REACTIVE_IGNORE,
                REACTIVE_IGNORE
            );
        }
    }

    function react(LogRecord calldata log) external vmOnly {

        if (log.topic_3 >= 0.001 ether) {
            bytes memory payload = abi.encodeWithSignature("callback(addres
s)", address(0));
            emit Callback(destinationChainId, callback, GAS_LIMIT, payload);
```

```
    }
  }
}
```

可能比较长，不要怕，我们一点点拆解

第一个点首先是consturctor函数，我们在部署时需要提供什么信息呢

address _service,Reactive Network在链上的服务地址，0x0000000000000000000000000000000000fffFfF

uint256 _originChainId, 源合约的链ID

uint256 _destinationChainId,目标合约的链ID

address _contract,源合约的部署地址

uint256 _topic_0 要监听的事件对topics[0]，比如这里我们要监听event Received(address from, address to, uint256 amount);，对应的哈希就是keccak256("Received(address,address,uint256)")

address _callback 目标合约的部署地址

当然，很多信息我们可以提前存到.env文件中，不需要每次都手动输入

第二个点是 31行的判断

!vm，还记得我们之前说的吗，虽然代码是一份，但实际上是运行在两处的两个实例，vm就是判断是否在Reactive VM上的标识，如何判断的也很有意思，通过确定地址0x0000000000000000000000000000000000fffFfF上是否有代码，代码则说明在Reactive Network上，如果！vm，就可以执行订阅外部链上的信息

第三个点是react函数

vmOnly说明只能在Reactive VM上运行，做的事情也很简单，如果第四个topics大于0.001ether，就发出callback虽然这里的签名是0地址，但实际上这里不会真正调用目标合约，而是告诉Reactive Network，要调用合约了，Reactive Network会将地址换成部署者的地址

好了，我们再回到目标合约

第一点，构造函数

参数只有一个，callback_sender，也就是callback是从哪个发出的，也就是**Reactive合约在对应链上的服务地址**，参见 https://dev.reactive.network/origins-and-destinations#callback-proxy-address

在sepolia上是 0xc9f36411C9897e7F959D99ffca2a0Ba7ee0D7bDA

第二点，函数的modifier

两个modifier进行两个验证

验证通道真实性，交易发起者是官方地址

验证交易发起人是“你”也就是部署Reactive合约的人

```
// SPDX-License-Identifier: GPL-2.0-or-later

pragma solidity >=0.8.0;

import '../../../lib/reactive-lib/src/abstract-base/AbstractCallback.sol';

contract BasicDemoL1Callback is AbstractCallback {
    event CallbackReceived(
        address indexed origin,
        address indexed sender,
        address indexed reactive_sender
    );

    constructor(address _callback_sender) AbstractCallback(_callback_send
er) payable {}

    function callback(address sender)
        external
        authorizedSenderOnly
        rvmIdOnly(sender)
    {
        emit CallbackReceived(
            tx.origin,
            msg.sender,
            sender
        );
```

```
    }
}
```

ok，到这里，这个最简单的例子你已经理解了，我们来实操一下，提前准备好Reactive网络上的代币，用于支付对应服务，参考 https://dev.reactive.network/reactive-mainnet#overview

Step1. 克隆项目后，首先配置环境变量，新建.env文件，配置完成

```
ORIGIN_RPC=源合约链的RPC
ORIGIN_CHAIN_ID=源合约链的ID
ORIGIN_PRIVATE_KEY=源合约链的私钥
DESTINATION_RPC=目标合约链的RPC
DESTINATION_CHAIN_ID=目标合约链的ID
DESTINATION_PRIVATE_KEY=目标合约链的私钥
REACTIVE_RPC=Reactive Network的RPC可以在 https://dev.reactive.network/reactive-mainnet#overview找到
REACTIVE_PRIVATE_KEY=你在reactive网络上的私钥
SYSTEM_CONTRACT_ADDR=Reactive Network上服务的合约地址，参考 https://dev.reactive.network/reactive-mainnet#overview
DESTINATION_CALLBACK_PROXY_ADDR=目标链对应的reactive服务合约地址，参考 https://dev.reactive.network/origins-and-destinations#callback-proxy-address
$CLIENT_WALLET$CLIENT_WALLET=你的钱包地址
```

Step2. 部署源合约：

运行前，先载入下环境变量

```
source .env
forge create --broadcast --rpc-url $ORIGIN_RPC --private-key $ORIGIN_PRIVATE_KEY src/demos/basic/BasicDemoL1Contract.sol:BasicDemoL1Contract
```

Step3. 部署目标合约：

```
forge create --broadcast --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY src/demos/basic/BasicDemoL1Callback.sol:BasicDemoL1Callback --value 0.02ether --constructor-args $DESTINATION_CALLBACK_PROXY_ADDR
```

Step4. 部署Reactive合约，注意将$ORIGIN_ADDR和$CALLBACK_ADDR替换成你前面部署的两个合约，或者你也可以先写在环境变量里

```
forge create --broadcast --rpc-url $REACTIVE_RPC --private-key $REACTIVE_PRIVATE_KEY src/demos/basic/BasicDemoReactiveContract.sol:BasicDemoReactiveContract --value 0.1ether --constructor-args $SYSTEM_CONTRACT_ADDR $ORIGIN_CHAIN_ID $DESTINATION_CHAIN_ID $ORIGIN_ADDR 0x8cabf31d2b1b11ba52dbb302817a3c9c83e4b2a5194d35121ab1354d69f6a4cb $CALLBACK_ADDR
```

Step5. 通过向源合约发送0.001ether，触发合约，注意替换$ORIGIN_ADDR

```
cast send $ORIGIN_ADDR --rpc-url $ORIGIN_RPC --private-key $ORIGIN_PRIVATE_KEY --value 0.001ether
```

到这，整个过程就完成了，我们到etherscan上检查下目标合约

可以看到，目标合约发出了CallbackReceived Event，其中：

origin是回调代理的地址，本质上是这个账户作为“用户”触发了合约

sender是Sepolia网络上Reactive合约服务的地址

reactive_sender是“你”

OK，恭喜你完成了第一个小实例，接下来我们进入第二个案例，利用Reactive合约执行Uniswap V2止损，这里我们更多的关注Reactive合约的触发，涉及ERC-20和Uniswap V2的知识，大家可以从SpeedRunEthurun或Uniswap v2 Developer Course - Cyfrin Updraft进行了解

Part3. Uniswap V2 止损单触发

在这个案例中，我们要监控一个代币对的价格，当价格抵达我们设计的阈值，自动触发帮助我们买入或卖出

首先依然要配置好环境变量，为了完整演示整个流程，我们首先要准备好Sepolia ETH和IReact代币，紧接着，我们要模拟代币对（部署两个代币，部署时合约会自动给部署者铸造100个代币）和Uniswap V2的交易对合约（用于代币的交换）

```
forge create --broadcast --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY src/demos/uniswap-v2-stop-order/UniswapDemoToken.sol:UniswapDemoToken --constructor-args TK1 TK1
```

将部署的合约存到环境变量里

export TK1=合约的地址

```
forge create --broadcast --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY src/demos/uniswap-v2-stop-order/UniswapDemoToken.sol:UniswapDemoToken --constructor-args TK2 TK2
```

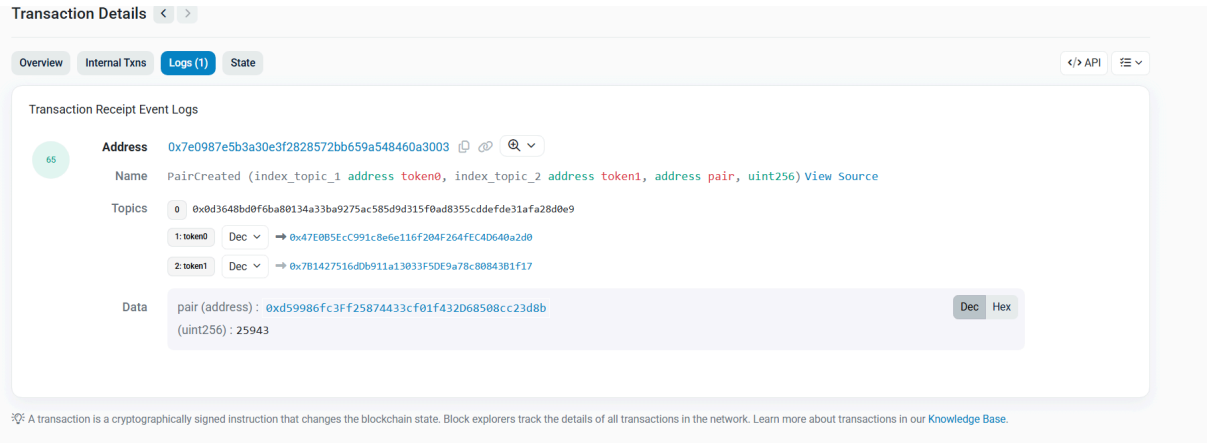
同样的，export TK2=合约的地址

当然，你也可以写到.env文件里

接下来，创建Uniswap交易对，在Sepolia上，Uniswap V2 Factory 合约 0x7E0987E5b3a30e3f2828572Bb659A548460a3003

```
cast send 0x7E0987E5b3a30e3f2828572Bb659A548460a3003 'createPair(address,address)' --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY $TK1 $TK2
```

完成后，要通过交易Hash到etherscan上找到对应的交易对



类似上图，其中的pair地址就是交易对地址

将地址存到环境变量里

export UNISWAP_V2_PAIR_ADDR=你的合约地址

接下来，部署目标合约，也就是最终会被调用的合约，看看构造函数，需要传入目标网络reactive服务合约地址和Uniswap router合约地址，sepolia上为0xC532a74256D3Db42D0Bf7a0400fEFDbad7694008

```
forge create --broadcast --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY src/demos/uniswap-v2-stop-order/UniswapDemoStopOrderCallback.sol:UniswapDemoStopOrderCallback --value 0.01ether --constructor-args $DESTINATION_CALLBACK_PROXY_ADDR $UNISWAP_V2_PAIR_ADDR
```

同样，把合约地址存到环境变量里

export CALLBACK_ADDR=你的地址

好了，接下来要开始准备交易对合约，对Uniswap V2了解的同学知道，一个交易对的正常运行，需要提供一定的流动性，我们将流动性添加到资金池中

```
cast send $TK1 'transfer(address,uint256)' --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY $UNISWAP_V2_PAIR_ADDR 1e18
cast send $TK2 'transfer(address,uint256)' --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY $UNISWAP_V2_PAIR_ADDR 1e18
cast send $UNISWAP_V2_PAIR_ADDR 'mint(address)' --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY $CLIENT_WALLET
```

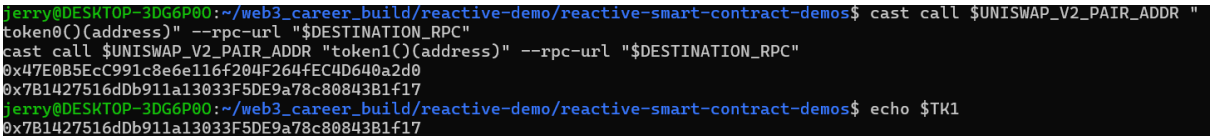
现在，流动性池也准备好了，检查一下

```
cast call $UNISWAP_V2_PAIR_ADDR "getReserves()(uint112,uint112,uint32)" --rpc-url "$DESTINATION_RPC"
```

需要说明的是，token0和token1取决于你的代币地址相对大小，**需要提前检查确认下token0和token1的地址，避免之后搞错**

```
cast call $UNISWAP_V2_PAIR_ADDR "token0()(address)" --rpc-url "$DESTINATION_RPC"
```

```
cast call $UNISWAP_V2_PAIR_ADDR "token1()(address)" --rpc-url "$DESTI
NATION_RPC"
```



可以看到，我这里token0是TK1，token1是TK2

接下来，就要部署最核心的Reactive合约了，

```
forge create --broadcast --rpc-url $REACTIVE_RPC --private-key $REACTI
VE_PRIVATE_KEY src/demos/uniswap-v2-stop-order/UniswapDemoStopOr
derReactive.sol:UniswapDemoStopOrderReactive --value 0.01ether --const
ructor-args $UNISWAP_V2_PAIR_ADDR $CALLBACK_ADDR $CLIENT_WALL
ET $DIRECTION_BOOLEAN $EXCHANGE_RATE_DENOMINATOR $EXCHAN
GE_RATE_NUMERATOR
```

最后三个参数可以提前在环境参数里指定，可以在命令行中直接替换

DIRECTION_BOOLEAN： true 表示卖出 token0 并买入 token1； false 表示反向操作

EXCHANGE_RATE_DENOMINATOR 和 EXCHANGE_RATE_NUMERATOR：汇率阈值，以整数表示。 EXCHANGE_RATE_DENOMINATOR 设置为 1000 ，EXCHANGE_RATE_NUMERATOR 设置为 1234，表示汇率阈值为1.234

这里我分别设置为 true 1000 900，也就是说，但token0降价到只要0.9个token1就可以兑换时，我们要卖出token0止损

下一步，因为你需要callback合约帮你操作代币兑换，你需要给合约一些代币的使用权，也就是approve()，这里我们给它一个代币

```
cast send $TK1 'approve(address,uint256)' --rpc-url $DESTINATION_RPC -
-private-key $DESTINATION_PRIVATE_KEY $CALLBACK_ADDR 1e18
cast send $TK2 'approve(address,uint256)' --rpc-url $DESTINATION_RPC
--private-key $DESTINATION_PRIVATE_KEY $CALLBACK_ADDR 1e18
```

好了，接下来只要代币间比值达到我们的阈值，理论上就可以触发交易，合约会卖出token0，买入token1

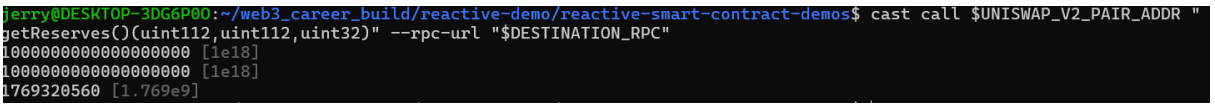
接下来，我们需要人为直接操作汇率，原理也很简单，向池子内投入更多的token1，拿出少量的token0，这个时候token0：token1就会减少，也就是1个token0能换更少的token1，也就是以token1计价，token0相对便宜了

为了实现这个操作，需要两个步骤

第一步给pair发一些token0，我这里是TK2

```
cast send $TK1 'transfer(address,uint256)' --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY $UNISWAP_V2_PAIR_ADDR 1e18
```

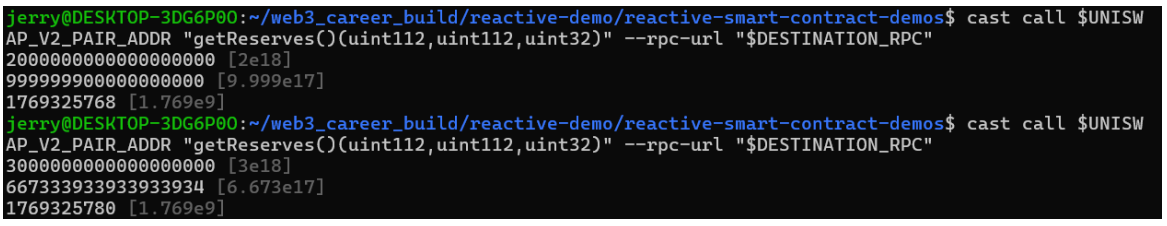
此时理论上应该有2e12个token0，但因为此时没有执行swap，所以此时检查流动性池会发现没有任何变化



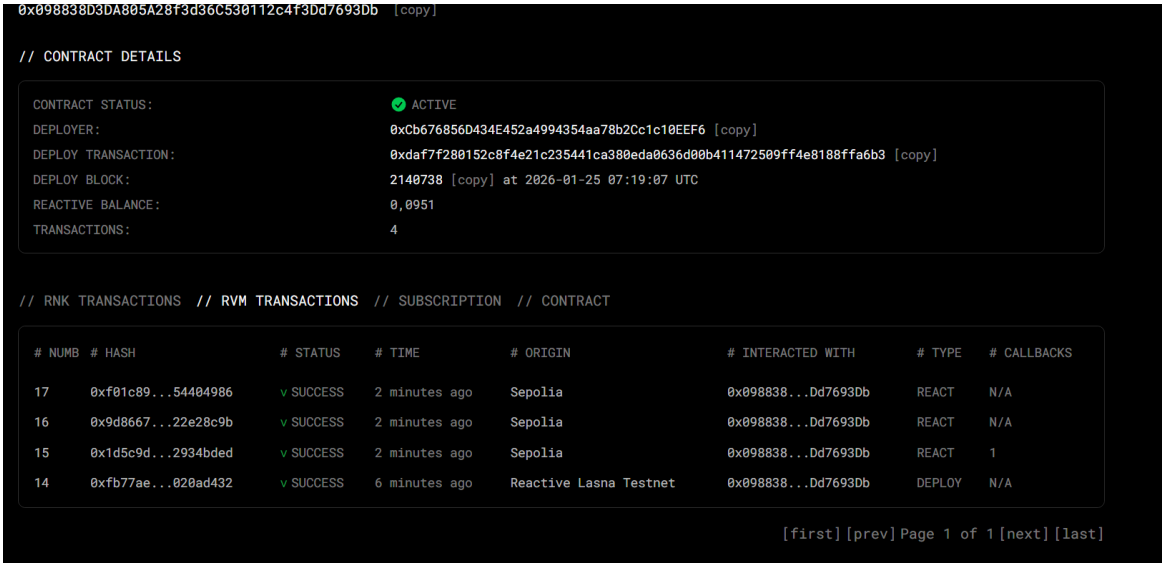
我们再做一次“吃亏”的交换swap

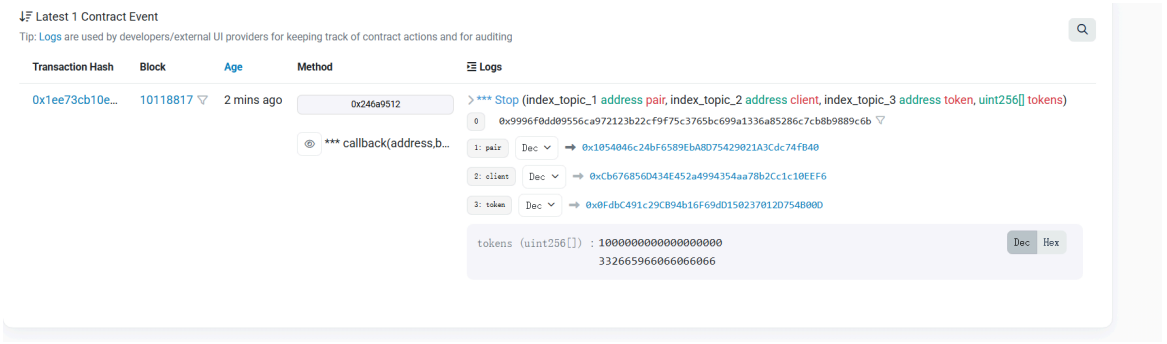
```
cast send $UNISWAP_V2_PAIR_ADDR 'swap(uint,uint,address,bytes calldata)' --rpc-url $DESTINATION_RPC --private-key $DESTINATION_PRIVATE_KEY 0 1e11 $CLIENT_WALLET "0x"
```

我们拿走0个token0和1e11个token1，此时，token0是2e12，token1是9.9e17，比值小于我们的阈值，因此触发了Reactive合约，swap后立马查看流动性，然后稍等再查看一次，可以看到交易自动发生了



我们可以在 https://lasna.reactscan.net/ 上搜索我们的Reactive合约地址，找到对应的触发交易，或者通过被触发的callback合约，看到输出的log（你看到的结果可能和我的不一样，因为我在尝试的时候对流动性做了很多更改）





到此，合约被完美触发了，但背后可能还有一些问题有待我们去探讨了解，一些细节必须要深入代码才能理解，这部分就当作大家的自主学习内容了

参考链接：

Introduction | Reactive Network

https://github.com/Reactive-Network/reactive-smart-contract-demos