分账与空头

一、分账

分账就是按照一定比例分钱财。在现实中，经常会有“分赃不均”的事情发生；而在区块链的世界里，`Code is Law`，我们可以事先把每个人应分的比例写在智能合约中，获得收入后，再由智能合约来进行分账。

分账合约

分账合约(PaymentSplit)具有以下几个特点：

1. 在创建合约时定好分账受益人 `payees` 和每人的份额 `shares` 。
2. 份额可以是相等，也可以是其他任意比例。
3. 在该合约收到的所有 `ETH` 中，每个受益人将能够提取与其分配的份额成比例的金额。
4. 分账合约遵循 `Pull Payment` 模式，付款不会自动转入账户，而是保存在此合约中。受益人通过调用 `release()` 函数触发实际转账。

- *// SPDX-License-Identifier: MIT*
  pragma solidity ^0.8.4;

  */***
  分账合约
  @dev 这个合约会把收到的ETH按事先定好的份额分给几个账户。收到ETH会存在分账合约中，需要每个受益人调用release()函数来领取。
  **/*
  contract PaymentSplit{

事件

分账合约中共有 `3` 个事件：

- `PayeeAdded`：增加受益人事件。
- `PaymentReleased`：受益人提款事件。
- `PaymentReceived`：分账合约收款事件。

```
    // 事件
    event PayeeAdded(address account, uint256 shares); // 增加受益人事件
    event PaymentReleased(address to, uint256 amount); // 受益人提款事件
    event PaymentReceived(address from, uint256 amount); // 合约收款事件
```

状态变量

分账合约中共有 `5` 个状态变量，用来记录受益地址、份额、支付出去的 `ETH` 等变量：

- `totalShares`：总份额，为 `shares` 的和。
- `totalReleased`：从分账合约向受益人支付出去的 `ETH`，为 `released` 的和。
- `payees`： `address` 数组，记录受益人地址
- `shares`： `address` 到 `uint256` 的映射，记录每个受益人的份额。
- `released`： `address` 到 `uint256` 的映射，记录分账合约支付给每个受益人的金额。

```
    uint256 public totalShares; // 总份额
    uint256 public totalReleased; // 总支付

    mapping(address => uint256) public shares; // 每个受益人的份额
```

```
    mapping(address => uint256) public released; // 支付给每个受益人的金额
    address[] public payees; // 受益人数组
```

函数

分账合约中共有 `6` 个函数：

- 构造函数：始化受益人数组 `_payees` 和分账份额数组 `_shares`，其中数组长度不能为0，两个数组长度要相等。_shares中元素要大于0，_payees中地址不能为0地址且不能有重复地址。
- `receive()`：回调函数，在分账合约收到 `ETH` 时释放 `PaymentReceived` 事件。
- `release()`：分账函数，为有效受益人地址 `_account` 分配相应的 `ETH`。任何人都可以触发这个函数，但 `ETH` 会转给受益人地址 `account`。调用了releasable()函数。
- `releasable()`：计算一个受益人地址应领取的 `ETH`。调用了 `pendingPayment()` 函数。
- `pendingPayment()`：根据受益人地址 `_account`,分账合约总收入 `_totalReceived` 和该地址已领取的钱 `_alreadyReleased`，计算该受益人现在应分的 `ETH`。
- `_addPayee()`：新增受益人函数及其份额函数。在合约初始化的时候被调用，之后不能修改。

```
    /**
     * @dev 初始化受益人数组_payees和分账份额数组_shares
     * 数组长度不能为0，两个数组长度要相等。_shares中元素要大于0，_payees中地址不能为0地址且不能有重复地址
     */
    constructor(address[] memory _payees, uint256[] memory _shares) payable {
        // 检查_payees和_shares数组长度相同，且不为0
        require(_payees.length == _shares.length, "PaymentSplitter: payees and shares length mismatch");
        require(_payees.length  0, "PaymentSplitter: no payees");
        // 调用_addPayee，更新受益人地址payees、受益人份额shares和总份额totalShares
        for (uint256 i = 0; i < _payees.length; i++) {
            _addPayee(_payees[i], _shares[i]);
        }
    }

    /**
     * @dev 回调函数，收到ETH释放PaymentReceived事件
     */
    receive() external payable virtual {
        emit PaymentReceived(msg.sender, msg.value);
    }

```

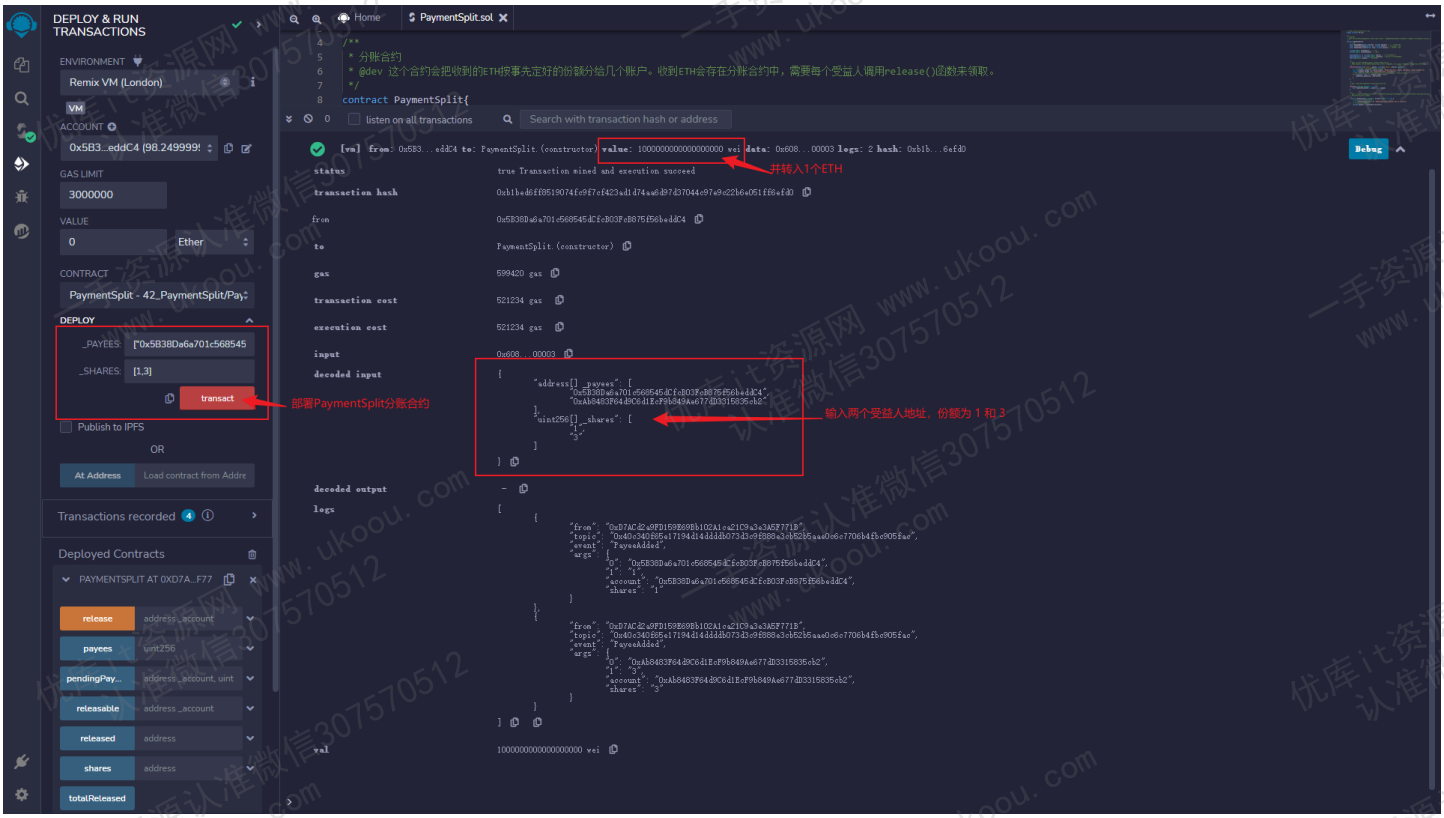
```
22      /**
23       * @dev 为有效受益人地址_account分帐，相应的ETH直接发送到受益人地址。任何人都可以触发这个函数，但钱会打给account地址。
24       * 调用了releasable()函数。
25       */
26      function release(address payable _account) public virtual {
27          // account必须是有效受益人
28          require(shares[_account]  0, "PaymentSplitter: account has no shares");
29          // 计算account应得的eth
30          uint256 payment = releasable(_account);
31          // 应得的eth不能为0
32          require(payment != 0, "PaymentSplitter: account is not due payment");
33          // 更新总支付totalReleased和支付给每个受益人的金额released
34          totalReleased += payment;
35          released[_account] += payment;
36          // 转账
37          _account.transfer(payment);
38          emit PaymentReleased(_account, payment);
39      }
40
41      /**
42       * @dev 计算一个账户能够领取的eth。
43       * 调用了pendingPayment()函数。
44       */
45      function releasable(address _account) public view returns (uint256) {
46          // 计算分账合约总收入totalReceived
47          uint256 totalReceived = address(this).balance + totalReleased;
48          // 调用_pendingPayment计算account应得的ETH
49          return pendingPayment(_account, totalReceived, released[_account]);
50      }
51
52      /**
53       * @dev 根据受益人地址_account，分账合约总收入_totalReceived和该地址已领取的钱_alreadyReleased，计算该受益人现在应分的ETH。
54       */
55      function pendingPayment(
56          address _account,
57          uint256 _totalReceived,
58          uint256 _alreadyReleased
59      ) public view returns (uint256) {
60          // account应得的ETH = 总应得ETH - 已领到的ETH
61          return (_totalReceived * shares[_account]) / totalShares - _alreadyReleased;
62      }
63
64      /**
```

```
65      * @dev 新增受益人_account以及对应的份额_accountShares。只能在构造器中被调用，不能修改。
66      */
67     function _addPayee(address _account, uint256 _accountShares) private {
68         // 检查_account不为0地址
69         require(_account != address(0), "PaymentSplitter: account is the zero address");
70         // 检查_accountShares不为0
71         require(_accountShares  0, "PaymentSplitter: shares are 0");
72         // 检查_account不重复
73         require(shares[_account] == 0, "PaymentSplitter: account already has shares");
74         // 更新payees，shares和totalShares
75         payees.push(_account);
76         shares[_account] = _accountShares;
77         totalShares += _accountShares;
78         // 释放增加受益人事件
79         emit PayeeAdded(_account, _accountShares);
80     }
```

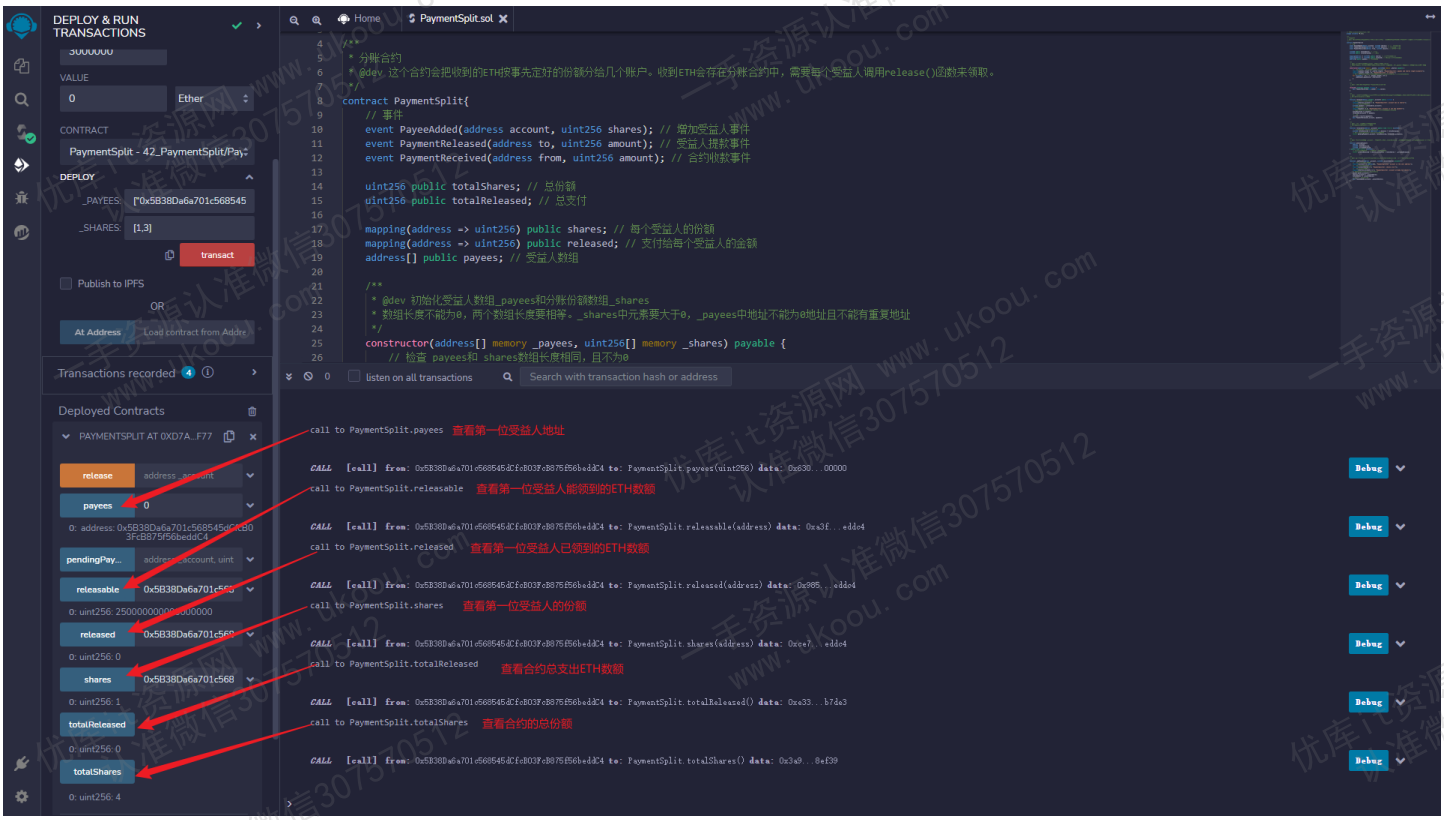
Remix 演示

5. 部署 PaymentSplit 分账合约，并转入 1 ETH

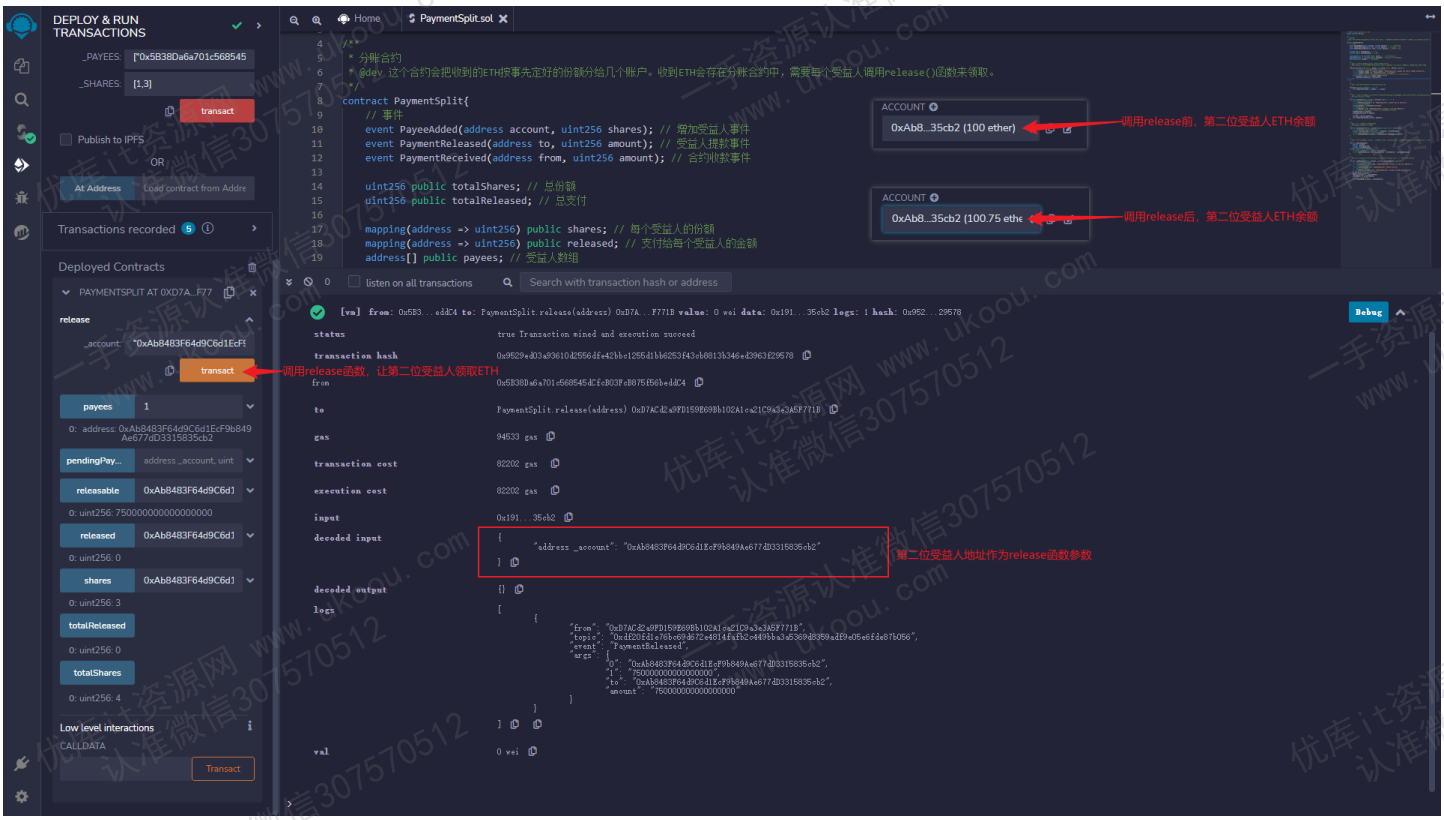
在构造函数中，输入两个受益人地址，份额为 1 和 3 。



6. 查看受益人地址、份额、应分到的 ETH



7. 函数领取 ETH



8. 查看总支出、受益人余额、应分到的 ETH 的变化

总结

这一讲，我们介绍了分账合约。在区块链的世界里，`Code is Law`，我们可以事先把每个人应分的比例写在智能合约中，获得收入后，由智能合约来进行分账，避免事后“分赃不均”。

二、空投 Airdrop

在币圈，最开心的一件事就是领空投，空手套白狼。这一讲，我们将学习如何使用智能合约发送`ERC20`代币空投。

空投是币圈中一种营销策略，项目方将代币免费发放给特定用户群体。为了拿到空投资格，用户通常需要完成一些简单的任务，如测试产品、分享新闻、介绍朋友等。项目方通过空投可以获得种子用户，而用户可以获得一笔财富，两全其美。

因为每次接收空投的用户很多，项目方不可能一笔一笔的转账。利用智能合约批量发放`ERC20`代币，可以显著提高空投效率。

空投代币合约

`Airdrop`空投合约逻辑非常简单：利用循环，一笔交易将`ERC20`代币发送给多个地址。合约中包含两个函数

- `getSum()` 函数：返回 `uint` 数组的和。

```
// 数组求和函数
function getSum(uint256[] calldata _arr) public pure returns(uint sum)
{
    for(uint i = 0; i < _arr.length; i++)
        sum = sum + _arr[i];
}
```

- `multiTransferToken()` 函数：发送 `ERC20` 代币空投，包含 `3` 个参数：
  - `_token`：代币合约地址（`address` 类型）
  - `_addresses`：接收空投的用户地址数组（`address[]` 类型）
  - `_amounts`：空投数量数组，对应 `_addresses` 里每个地址的数量（`uint[]` 类型）
  - 该函数有两个检查：第一个 `require` 检查了 `_addresses` 和 `_amounts` 两个数组长度是否相等；第二个 `require` 检查了空投合约的授权额度大于要空投的代币数量总和。

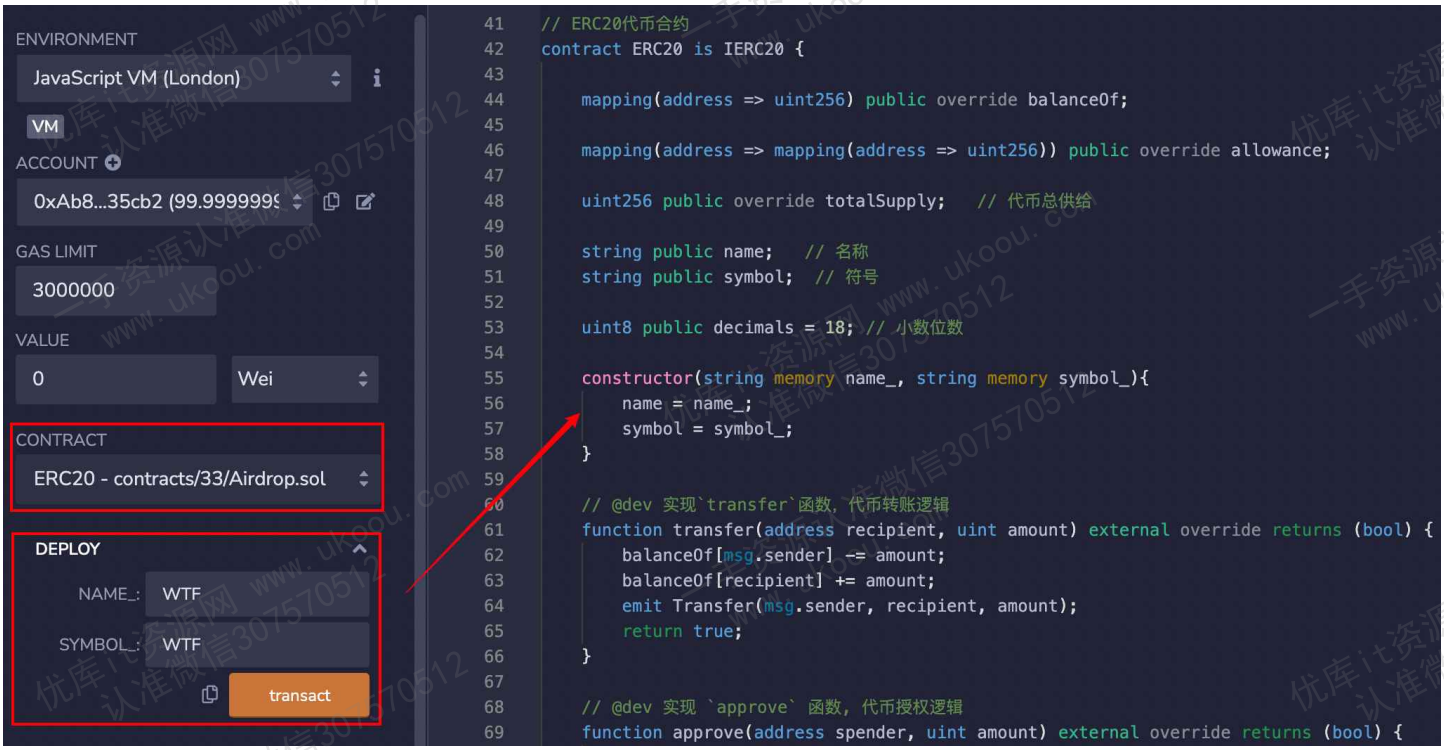
```
/// @notice 向多个地址转账ERC20代币，使用前需要先授权
///
/// @param _token 转账的ERC20代币地址
/// @param _addresses 空投地址数组
/// @param _amounts 代币数量数组（每个地址的空投数量）
function multiTransferToken(
    address _token,
    address[] calldata _addresses,
    uint256[] calldata _amounts
    ) external {
    // 检查：_addresses和_amounts数组的长度相等
    require(_addresses.length == _amounts.length, "Lengths of Addresses and Amounts NOT EQUAL");
    IERC20 token = IERC20(_token); // 声明IERC合约变量
    uint _amountSum = getSum(_amounts); // 计算空投代币总量
    // 检查：授权代币数量 >= 空投代币总量
    require(token.allowance(msg.sender, address(this)) >= _amountSum, "Need Approve ERC20 token");

    // for循环，利用transferFrom函数发送空投
    for (uint8 i; i < _addresses.length; i++) {
        token.transferFrom(msg.sender, _addresses[i], _amounts[i]);
    }
}
```

- `multiTransferETH()` 函数：发送 `ETH` 空投，包含 `2` 个参数：
  - `_addresses`：接收空投的用户地址数组（`address[]` 类型）
  - `_amounts`：空投数量数组，对应 `_addresses` 里每个地址的数量（`uint[]` 类型）

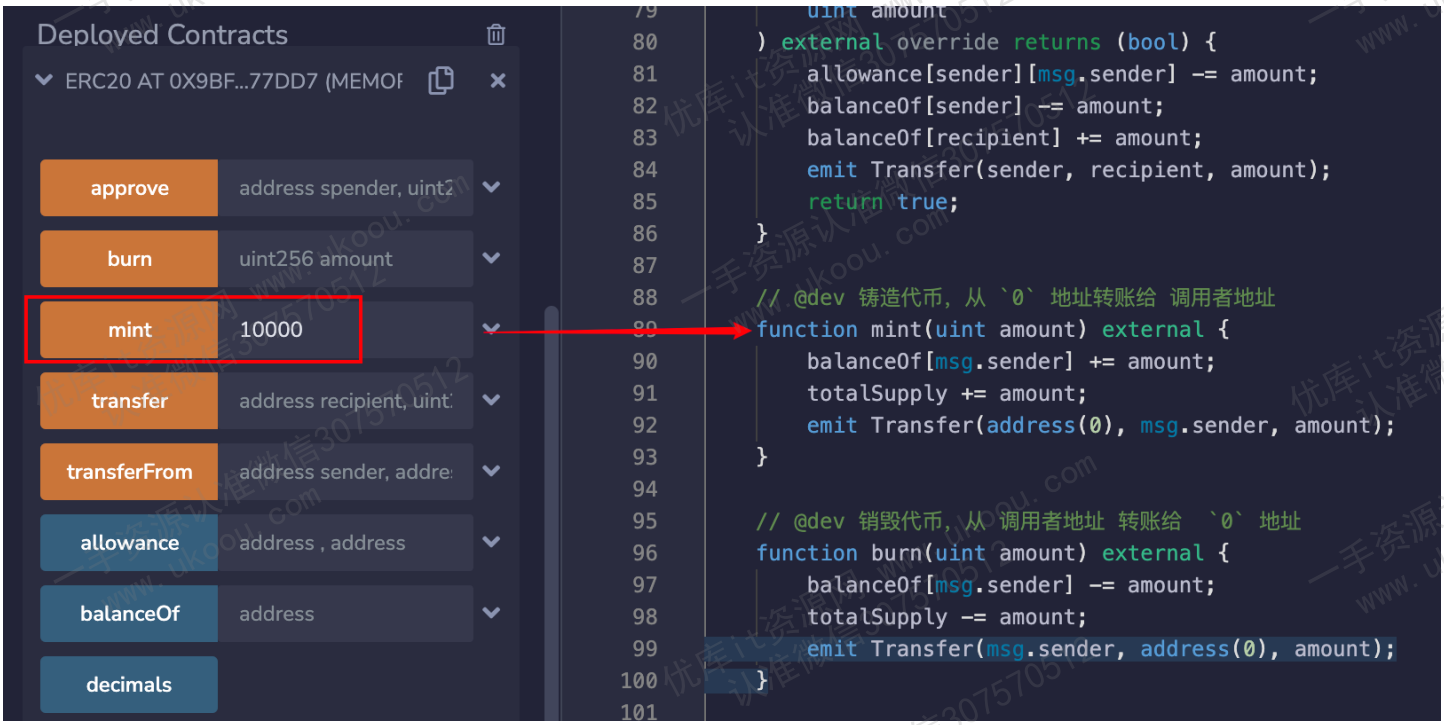
```
/// 向多个地址转账ETH
function multiTransferETH(
    address payable[] calldata _addresses,
    uint256[] calldata _amounts
) public payable {
    // 检查：_addresses和_amounts数组的长度相等
    require(_addresses.length == _amounts.length, "Lengths of Addresses and Amounts NOT EQUAL");
    uint _amountSum = getSum(_amounts); // 计算空投ETH总量
    // 检查转入ETH等于空投总量
    require(msg.value == _amountSum, "Transfer amount error");
    // for循环，利用transfer函数发送ETH
    for (uint256 i = 0; i < _addresses.length; i++) {
        _addresses[i].transfer(_amounts[i]);
    }
}
```

空投实践

1. 部署 `ERC20` 代币合约，并给自己 `mint` 10000 单位代币。

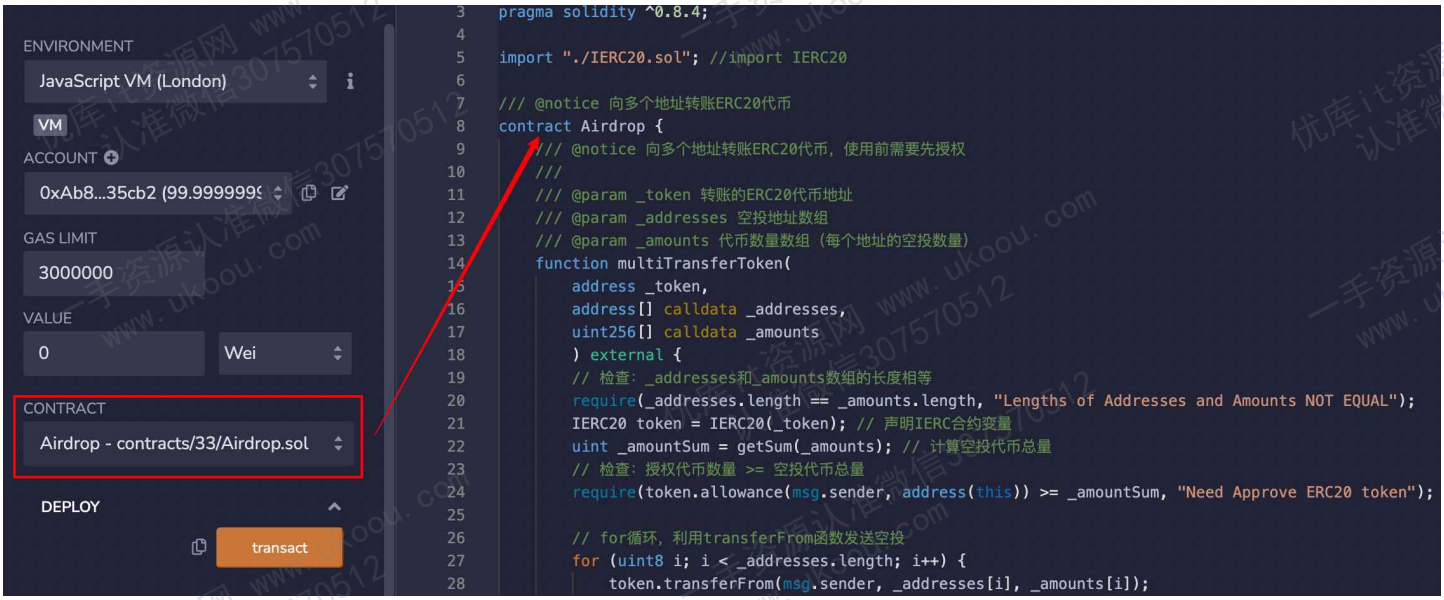



```solidity
// ERC20代币合约
contract ERC20 is IERC20 {

    mapping(address => uint256) public override balanceOf;

    mapping(address => mapping(address => uint256)) public override allowance;

    uint256 public override totalSupply;   // 代币总供给

    string public name;   // 名称
    string public symbol;  // 符号

    uint8 public decimals = 18; // 小数位数

    constructor(string memory name_, string memory symbol_){
        name = name_;
        symbol = symbol_;
    }

    // @dev 实现`transfer`函数，代币转账逻辑
    function transfer(address recipient, uint amount) external override returns (bool) {
        balanceOf[msg.sender] -= amount;
        balanceOf[recipient] += amount;
        emit Transfer(msg.sender, recipient, amount);
        return true;
    }

    // @dev 实现 `approve` 函数，代币授权逻辑
    function approve(address spender, uint amount) external override returns (bool) {
```




```solidity
        uint amount
    ) external override returns (bool) {
        allowance[sender][msg.sender] -= amount;
        balanceOf[sender] -= amount;
        balanceOf[recipient] += amount;
        emit Transfer(sender, recipient, amount);
        return true;
    }

    // @dev 铸造代币，从 `0` 地址转账给 调用者地址
    function mint(uint amount) external {
        balanceOf[msg.sender] += amount;
        totalSupply += amount;
        emit Transfer(address(0), msg.sender, amount);
    }

    // @dev 销毁代币，从 调用者地址 转账给 `0` 地址
    function burn(uint amount) external {
        balanceOf[msg.sender] -= amount;
        totalSupply -= amount;
        emit Transfer(msg.sender, address(0), amount);
    }

```

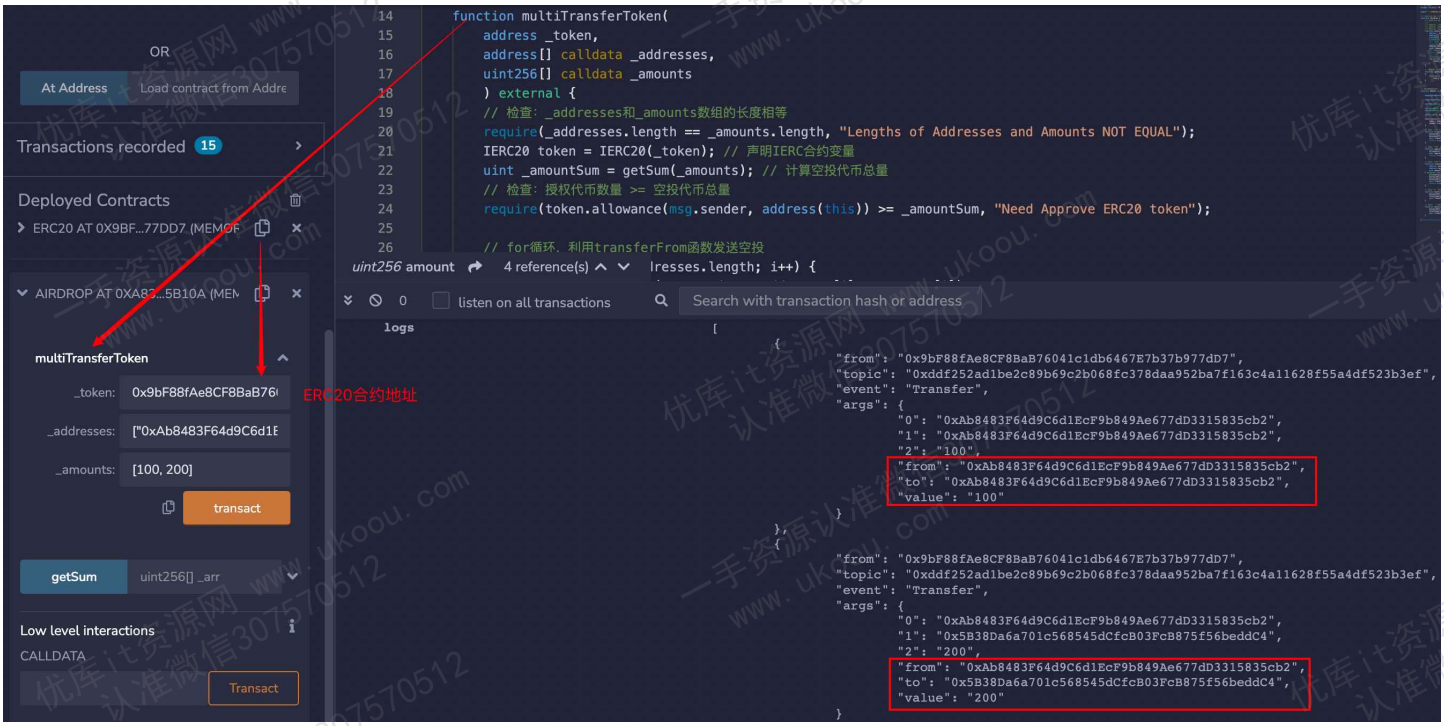
1. 部署 `Airdrop` 空投合约。



1. 利用 `ERC20` 代币合约中的 `approve()` 函数，给 `Airdrop` 空投合约授权10000 单位代币。

1. 执行 `Airdrop` 合约的 `multiTransferToken()` 函数进行空投， `_token` 填 `ERC20` 代币地址， `_addresses` 和 `_amounts` 按照以下填写

```
// _addresses填写
["0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
"0x5B38Da6a701c568545dCfcB03FcB875f56beddC4"]

// _amounts填写
[100, 200]
```



1. 利用 `ERC20` 合约的 `balanceOf()` 函数查询上面用户地址的代币余额，成功变为 `100` 和 `200`，空投成功！

总结

这一讲，我们介绍了如何使用 `solidity` 写 `ERC20` 代币空投合约，极大增加空投效率。我撸空投收获最大的一次是 `ENS` 空投，你们呢？