Solidity 智能合约开发 - 基础

前言

去年读研的时候上的 HKU 的 `<COMP7408 Distributed Ledger and Blockchain Technology>`，课程中学习了以太坊智能合约的开发，做了一个简单的图书管理 ĐApp，然后毕业设计也选择了基于 Ethereum 做了一个音乐版权应用，详见 Uright - 区块链音乐版权管理 ĐApp，对 Solidity 开发有一些基础了解。

后来工作后主要做联盟链和业务开发这一块，很久没有碰过合约，对于语法和底层一些概念都已经一知半解，正好最近做的项目是基于 EVM 的一条链，涉及了一些基本的存证、回检和迁移相关合约的开发，调试起来有些吃力，于是打算系统学习一下，梳理一下笔记成文章，敦促自己好好思考总结。

这系列文章也会收录在我的个人知识库项目 《区块链入门指南》中，希望在学习过程中不断完善。有兴趣的朋友也可以访问项目仓库参与贡献或提出建议。

本文为系列第一篇，主要涉及 Solidity 基础知识。

智能合约 与 Solidity 语言

智能合约是运行在链上的程序，合约开发者可以通过智能合约实现与链上资产/数据进行交互，用户可以通过自己的链上账户来调用合约，访问资产与数据。因为区块链保留区块历史记录的链式结构、去中心化、不可篡改等特征，智能合约相比传统应用来说能更公正、透明。

然而，因为智能合约需要与链进行交互，部署、数据写入等操作都会消耗一定费用，数据存储与变更成本也比较高，因此在设计合约时需要着重考虑资源的消耗。此外，常规智能合约一经部署就无法进行修改，因此，合约设计时也需要多考虑其安全性、可升级性与拓展性。

Solidity 是一门面向合约的、为实现智能合约而创建的高级编程语言，在 EVM 虚拟机上运行，语法整体类似于 Javascript，是目前最流行的智能合约语言，也是入门区块链与 Web3 所必须掌握的语言。针对上述的一些合约编写的问题，Solidity 也都有相对完善的解决方案支持，后续会详细讲解。

开发/调试工具

与常规编程语言不同，Solidity 智能合约的开发往往无法直接通过一个 IDE 或本地环境进行方便的调试，而是需要与一个链上节点进行交互。开发调试往往也不会直接与主网（即真实资产、数据与业务所在的链）进行交互，否则需要承担高额手续费。目前开发调试主要有以下几种方式与框架：

1. Remix IDE。通过 Ethereum 官方提供的基于浏览器的 Remix 开发工具进行调试，Remix 会提供完整的 IDE、编译工具、部署调试的测试节点环境、账户等，可以很方便地进行测试，这是我学习使用时用的最多的工具。Remix 还可以通过 MetaMask 插件与测试网、主网进行直接交互，部分生产环境也会使用它进行编译部署。
2. Truffle。Truffle 是一个非常流行的 Javascript 的 Solidity 合约开发框架，提供了完整的开发、测试、调试工具链，可以与本地或远程网络进行交互。
3. Brownie。Brownie 是一个基于 Python 的 Solidity 合约开发框架，以简洁的 Python 语法为调试和测试提供了便捷的工具链。
4. Hardhat。Hardhat 是另一个基于 Javascript 的开发框架，提供了非常丰富的插件系统，适合开发复杂的合约项目。

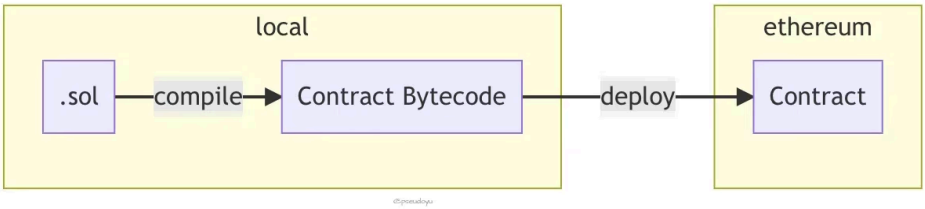
除了开发框架外，更好地进行 Solidity 还需要熟悉一些工具：

1. Remix IDE 对于语法提示等并不完善，因此，可以使用 Visual Studio Code 配合 Solidity 进行编写，有更好的体验。
2. MetaMask。一个常用的钱包应用，开发过程中可以通过浏览器插件与测试网、主网进行交互，方便开发者进行调试。

3. Ganache。Ganache 是一个开源的虚拟本地节点，提供了一个虚拟链网络，可以通过各类 Web3.js、Remix 或一些框架工具与之交互，适合有一定规模的项目进行本地调试与测试。
4. Infura。Infura 是一个 IaaS（Infrastructure as a Service）产品，我们可以申请自己的 Ethereum 节点，通过 Infura 提供的 API 进行交互，可以很方便地进行调试，也更接近生产环境。
5. OpenZeppelin。OpenZeppelin 提供了非常多的合约开发库与应用，能兼顾安全、稳定的同时给予开发者更好的开发体验，降低合约开发成本。

合约编译/部署

Solidity 合约是以 `.sol` 为后缀的文件，无法直接执行，需要编译为 EVM（Ethereum Virtual Machine）可识别的字节码才能在链上运行。



编译完成后，由合约账户进行部署到链上，其他账户可通过钱包与合约进行交互，实现链上业务逻辑。

核心语法

经过上文，我们对 Solidity 的开发、调试与部署有了一定了解。接下来我们就具体学习一下 Solidity 的核心语法。

数据类型

与我们常见的编程语言类似，Solidity 有一些内置数据类型。

基本数据类型

- `boolean`，布尔类型有 `true` 和 `false` 两种类型，可以通过 `bool public boo = true;` 来定义，默认值为 `false`
- `int`，整数类型，可以指定 `int8` 到 `int256`，默认为 `int256`，通过 `int public int = 0;` 来定义，默认值为 `0`，还可以通过 `type(int).min` 和 `type(int).max` 来查看类型最小和最大值
- `uint`，非负整数类型，可以指定 `uint8`、`uint16`、`uint256`，默认为 `uint256`，通过 `uint8 public u8 = 1;` 来定义，默认值为 `0`
- `address`，地址类型，可以通过 `address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;` 来定义，默认值为 `0x0000000000000000000000000000000000000000`

- `bytes`， `byte[]` 的缩写，分为固定大小数组和可变数组，通过 `bytes1 a = 0xb5;` 来定义

还有一些相对复杂的数据类型，我们单独进行讲解。

Enum

`Enum` 是枚举类型，可以通过以下语法来定义

```
enum Status {
    Unknown,
    Start,
    End,
    Pause
}
```

并通过以下语法来进行更新与初始化

```
// 实例化枚举类型
Status public status;

// 更新枚举值
function pause() public {
    status = Status.Pause;
}

// 初始化枚举值
function reset() public {
    delete status;
}
```

数组

数组是一种存储同类元素的有序集合，通过 `uint[] public arr;` 来进行定义，在定义时可以预先指定数组大小，如 `uint[10] public myFixedSizeArr;` 。

需要注意的是，我们可以在内存中创建数组（关于 `memory` 与 `storage` 等差异后续会详细讲解），但是必须固定大小，如 `uint[] memory a = new uint[](5);` 。

数组类型有一些基本操作方法，如下：

```
// 定义数组类型
uint[7] public arr;
```

```
// 添加数据
arr.push(7);

// 删除最后一个数据
arr.pop();

// 删除某个索引值数据
delete arr[1];

// 获取数组长度
uint len = arr.length;
```

mapping

`mapping` 是一种映射类型，使用 `mapping(keyType => valueType)` 来定义，其中键需要是内置类型，如 `bytes` 、 `string` 、 `string` 或合约类型，而值可以是任何类型，如嵌套 `mapping` 类型。需要注意的是， `mapping` 类型是不能被迭代遍历的，需要遍历则需要自行实现对应索引。

下面说明一下各类操作：

```
// 定义嵌套 mapping 类型
mapping(string => mapping(string => string)) nestedMap;

// 设置值
nestedMap[id][key] = "0707";

// 读取值
string value = nestedMap[id][key];

// 删除值
delete nestedMap[id][key];
```

Struct

`struct` 是结构类型，对于复杂业务，我们经常需要定义自己的结构，将关联的数据组合起来，可以在合约内进行定义

```
contract Struct {
    struct Data {
        string id;
        string hash;
    }
```

```
    Data public data;

    // 添加数据
    function create(string calldata _id) public {
        data = Data{id: _id, hash: "111222"};
    }

    // 更新数据
    function update(string _id) public {
        // 查询数据
        string id = data.id;

        // 更新
        data.hash = "222333"
    }
}
```

也可以单独文件定义所有需要的结构类型，由合约按需导入

```
// 'StructDeclaration.sol'

struct Data {
    string id;
    string hash;
}
```

```
// 'Struct.sol'

import "./StructDeclaration.sol"

contract Struct {
    Data public data;
}
```

变量/常量/ Immutable

变量是 Solidity 中可改变值的一种数据结构，分为以下三种：

- local 变量
- state 变量
- global 变量

其中， `local` 变量定义在方法中，而不会存储在链上，如 `string var = "Hello";`；而 `state` 变量在方法之外定义，会存储在链上，通过 `string public var;` 定义变量，写入值时会发送交易，而读取值则不会； `global` 变量则是提供了链信息的全局变量，如当前区块时间戳变量， `uint timestamp = block.timestamp;`，合约调用者地址变量， `address sender = msg.sender;` 等。

变量可以通过不同关键字进行声明，表示不同的存储位置。

- `storage`，会存储在链上
- `memory`，在内存中，只有方法被调用的时候才存在
- `calldata`，作为调用方法传入参数时存在

而常量是一种不可以改变值的变量，使用常量可以节约 gas 费用，我们可以通过 `string public constant MY_CONSTANT = "0707";` 来进行定义。 `immutable` 则是一种特殊的类型，它的值可以在 `constructor` 中初始化，但不可以再次改变。灵活使用这几种类型可以有效节省 gas 费并保障数据安全。

函数

在 Solidity 中，函数用来定义一些特定业务逻辑。

权限声明

函数分为不同的可见性，用户不同的关键字进行声明：

- `public`，任何合约都可调用
- `private`，只有定义了该方法的合约内部可调用
- `internal`，只有在继承合约可调用
- `external`，只有其他合约和账户可调用

查询数据的合约函数也有不同的声明方式：

- `view` 可以读取变量，但不能更改
- `pure` 不可以读也不可以修改

函数修饰符

`modifier` 函数修饰符可以在函数运行前/后被调用，主要用来进行权限控制、对输入参数进行校验以及防止重入攻击等。这三种功能修饰符可以通过以下语法定义：

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
}
```

```
    modifier validAddress(address _addr) {
        require(_addr != address(0), "Not valid address");
        _;
    }

    modifier noReentrancy() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }
```

使用函数修饰符则是需要在函数声明时添加对应修饰符，如：

```
    function changeOwner(address _newOwner) public onlyOwner validAddress(_newOwn
        owner = _newOwner;
    }

    function decrement(uint i) public noReentrancy {
        x -= i;

        if (i > 1) {
            decrement(i - 1);
        }
    }
```

函数选择器

当函数被调用时， `calldata` 的前四个字节要指定以确认调用哪个函数，被称为函数选择器。

```
    addr.call(abi.encodeWithSignature("transfer(address,uint256)", 0xSomeAddress,
```

上述代码 `abi.encodeWithSignature()` 返回值的前四个字节就是函数选择器。我们如果在执行前预先计算函数选择器的话可以节约一些 `gas` 费。

```
    contract FunctionSelector {
        function getSelector(string calldata _func) external pure returns (bytes4
            return bytes4(keccak256(bytes(_func)));
        }
    }
```

条件/循环结构

条件

Solidity 使用 `if`、`else if`、`else` 关键字来实现条件逻辑：

```
if (x < 10) {
    return 0;
} else if (x < 20) {
    return 1;
} else {
    return 2;
}
```

也可以使用简写形式：

```
x < 20 ? 1 : 2;
```

循环

Solidity 使用 `for`、`while`、`do while` 关键字来实现循环逻辑，但是因为后两者容易达到 `gas limit` 边界值，所以基本上不用。

```
for (uint i = 0; i < 10; i++) {
    // 业务逻辑
}
```

```
uint j;
while (j < 10) {
    j++;
}
```

合约

构造器

Solidity 的 `constructor` 可以在创建合约的时候执行，主要用来初始化

```
constructor(string memory _name) {
    name = _name;
}
```

如果合约之间存在继承关系，`constructor` 也会按照继承顺序。

接口

`Interface`，通过声明接口来进行合约交互，有以下要求：

- 不能实现任何方法
- 可以继承其他接口
- 的所有方法都必须声明为 `external`
- 不能声明构造方法
- 不能声明状态变量

接口用如下语法进行定义：

```
contract Counter {
    uint public count;

    function increment() external {
        count += 1;
    }
}

interface ICounter {
    function count() external view returns (uint);
    function increment() external;
}
```

调用则是通过

```
contract MyContract {
    function incrementCounter(address _counter) external {
        ICounter(_counter).increment();
    }

    function getCount(address _counter) external view returns (uint) {
        return ICounter(_counter).count();
    }
}
```

继承

Solidity 合约支持继承，且可以同时继承多个，使用 `is` 关键字。

函数可以进行重写，需要被继承的合约方法需要声明为 `virtual`，重写方法需要使用 `override` 关键字。

```
// 定义父合约 A
contract A {
    function foo() public pure virtual returns (string memory) {
        return "A";
    }
}

// B 合约继承 A 合约并重写函数
contract B is A {
    function foo() public pure virtual override returns (string memory) {
        return "B";
    }
}

// D 合约继承 B、C 合约并重写函数
contract D is B, C {
    function foo() public pure override(B, C) returns (string memory) {
        return super.foo();
    }
}
```

有几点需要注意的是，继承顺序会影响业务逻辑， `state` 状态变量是不可以被继承的。

如果子合约想调用父合约，除了直接调用外，还可以通过 `super` 关键字来调用，如下：

```
contract B is A {
    function foo() public virtual override {
        // 直接调用
        A.foo();
    }

    function bar() public virtual override {
        // 通过 super 关键字调用
        super.bar();
    }
}
```

合约创建

Solidity 中可以从另一个合约中使用 `new` 关键字来创建另一个合约

```
function create(address _owner, string memory _model) public {
    Car car = new Car(_owner, _model);
    cars.push(car);
```

```
}
```

而 solidity 0.8.0 后支持 create2 特性创建合约

```
function create2(address _owner, string memory _model, bytes32 _salt) public
    Car car = (new Car){salt: _salt}(_owner, _model);
    cars.push(car);
}
```

导入合约/外部库

复杂业务中，我们往往需要多个合约之间进行配合，这时候可以使用 import 关键字来导入合约，分为本地导入 import "./Foo.sol"; 与外部导入 import "https://github.com/owner/repo/blob/branch/path/to/Contract.sol"; 两种方式。

外部库和合约类似，但不能声明状态变量，也不能发送资产。如果库的所有方法都是 internal 的话会被嵌入合约，如果非 internal，需要提前部署库并且链接起来。

```
library SafeMath {
    function add(uint x, uint y) internal pure returns (uint) {
        uint z = x + y;
        require(z >= x, "uint overflow");
        return z;
    }
}


contract TestSafeMath {
    using SafeMath for uint;
}
```

事件

事件机制是合约中非常重要的一个设计。事件允许将信息记录到区块链上，DApp 等应用可以通过监听事件数据来实现业务逻辑，存储成本很低。以下是一个简单的日志抛出机制：

```
// 定义事件
event Log(address indexed sender, string message);
event AnotherLog();

// 抛出事件
emit Log(msg.sender, "Hello World!");
emit Log(msg.sender, "Hello EVM!");
```

```
    emit AnotherLog();
```

定义事件时可以传入 `indexed` 属性，但最多三个，加了后可以对这个属性的参数进行过滤，`var event = myContract.transfer({value: ["99","100","101"]});` 。

错误处理

链上错误处理也是合约编写的重要环节。Solidity 可以通过以下几种方式抛出错误。

`require` 都是在执行前验证条件，不满足则抛出异常。

```
function testRequire(uint _i) public pure {
    require(_i > 10, "Input must be greater than 10");
}
```

`revert` 用来标记错误与进行回滚。

```
function testRevert(uint _i) public pure {
    if (_i <= 10) {
        revert("Input must be greater than 10");
    }
}
```

`assert` 要求一定要满足条件。

```
function testAssert() public view {
    assert(num == 0);
}
```

注意，在 Solidity 中，当出现错误时会回滚交易中发生的所有状态改变，包括所有的资产，账户，合约等。

`try / catch` 也可以捕捉错误，但只能捕捉来自外部函数调用和合约创建的错误。

```
event Log(string message);
event LogBytes(bytes data);

function tryCatchNewContract(address _owner) public {
    try new Foo(_owner) returns (Foo foo) {
        emit Log("Foo created");
    } catch Error(string memory reason) {
        emit Log(reason);
```

```
        } catch (bytes memory reason) {
            emit LogBytes(reason);
        }
    }
```

payable 关键字

我们可以通过声明 payable 关键字设置方法可从合约中接收 ether 。

```
// 地址类型可以声明 payable
address payable public owner;

constructor() payable {
    owner = payable(msg.sender);
}

// 方法声明 payable 来接收 Ether
function deposit() public payable {}
```

与 Ether 交互

与 Ether 交互是智能合约的重要应用场景，主要分为发送和接收两部分，分别有不同的方法实现。

发送

主要通过 transfer 、 send 与 call 方法实现，其中 call 优化了对重入攻击的防范，在实际应用场景中建议使用（但一般不用来调用其他函数）。

```
contract SendEther {
  function sendViaCall(address payable _to) public payable {
    (bool sent, bytes memory data) = _to.call{value: msg.value}("");
    require(sent, "Failed to send Ether");
  }
}
```

而如果需要调用另一个函数，则一般使用 delegatecall 。

```
contract B {
    uint public num;
    address public sender;
    uint public value;
```

```
    function setVars(uint _num) public payable {
        num = _num;
        sender = msg.sender;
        value = msg.value;
    }
}

contract A {
    uint public num;
    address public sender;
    uint public value;

    function setVars(address _contract, uint _num) public payable {
        (bool success, bytes memory data) = _contract.delegatecall(
            abi.encodeWithSignature("setVars(uint256)", _num)
        );
    }
}
```

接收

接收 Ether 主要用 receive() external payable 与 fallback() external payable 两种。

当一个不接受任何参数也不返回任何参数的函数、当 Ether 被发送至某个合约但 receive() 方法未实现或 msg.data 非空时，会调用 fallback() 方法。

```
contract ReceiveEther {

    // 当 msg.data 为空时
    receive() external payable {}

    // 当 msg.data 非空时
    fallback() external payable {}

    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Gas 费

在 EVM 中执行交易需要耗费 gas 费， gas spent 表示需要多少 gas 量， gas price 为 gas 的单位价格， Ether 和 Wei 是价格单位，1 ether == 1e18 wei。

合约会对 Gas 进行限制， `gas limit` 由发起交易的用户设置，最多花多少 gas， `block gas limit` ，由区块链网络决定，这个区块中最多允许多少 gas。

我们在合约开发中要尤其考虑尽量节约 gas 费，有以下几个常用技巧：

1. 使用 `calldata` 来替换 `memory`
2. 将状态变量载入内存
3. 使用 `i++` 而不是 `++i`
4. 缓存数组元素

```
function sumIfEvenAndLessThan99(uint[] calldata nums) external {
    uint _total = total;
    uint len = nums.length;

    for (uint i = 0; i < len; ++i) {
        uint num = nums[i];
        if (num % 2 == 0 && num < 99) {
            _total += num;
        }
    }

    total = _total;
}
```

总结

以上就是我们系列第一篇，Solidity 基础知识，后续文章会对其常见应用和实用编码技巧进行学习总结，欢迎大家持续关注。

参考资料

1. Solidity by Example
2. Ethereum 區塊鏈！智能合約(Smart Contract)與分散式網頁應用(dApp)入門
3. 区块链入门指南
4. Uright - 区块链音乐版权管理ĐApp