

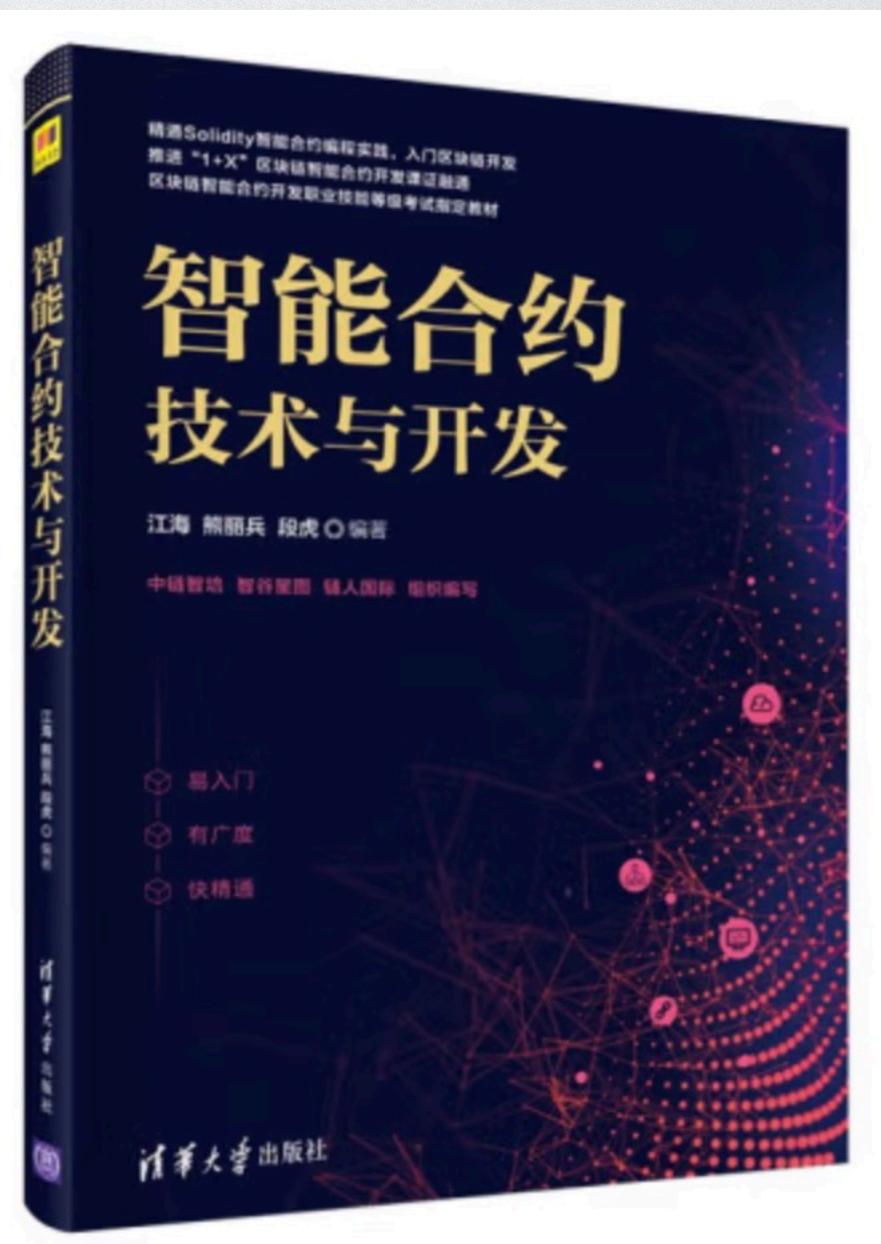
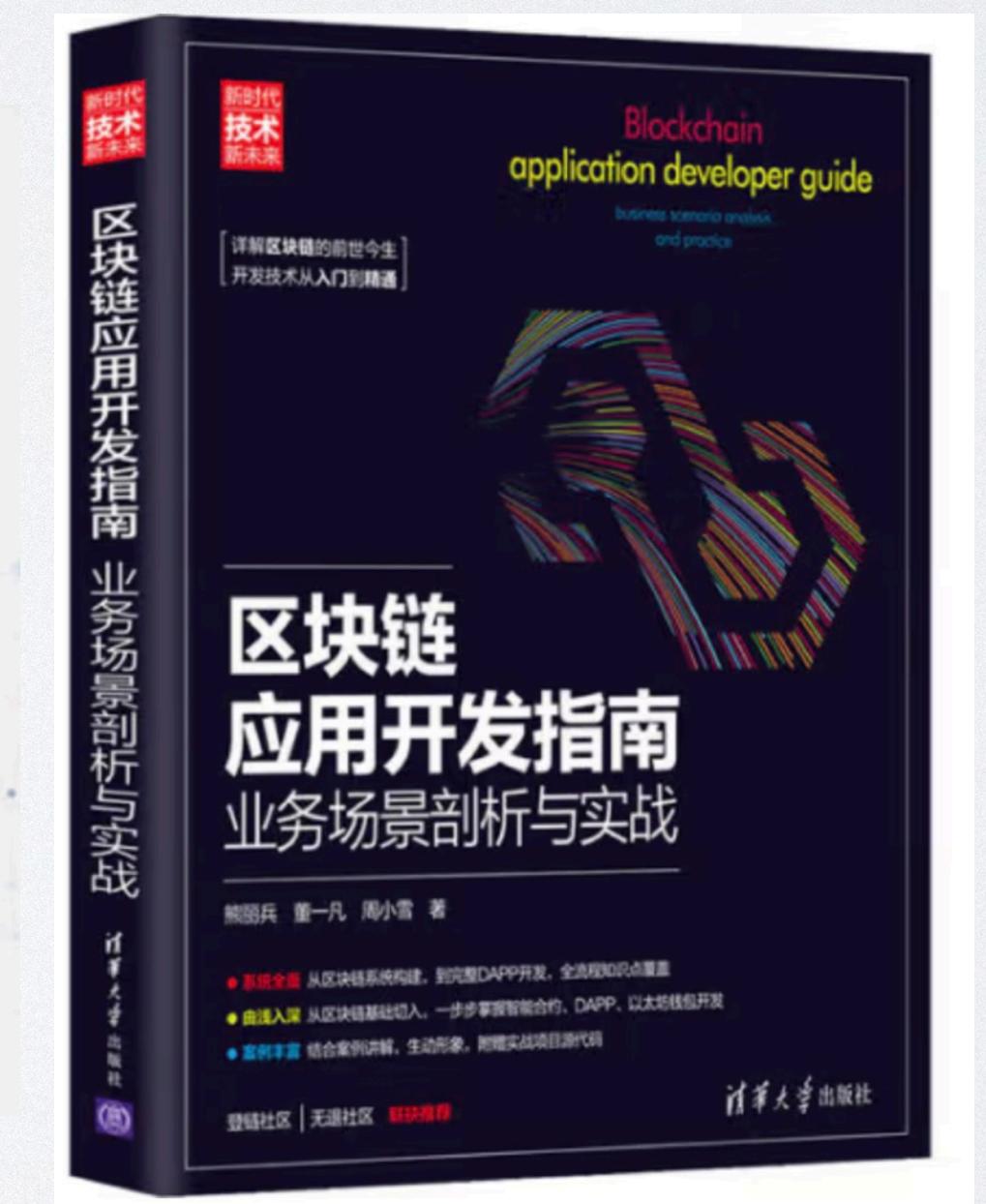
# 区块链集训营

## 二期

登链社区 - Tiny熊

# ABOUT ME

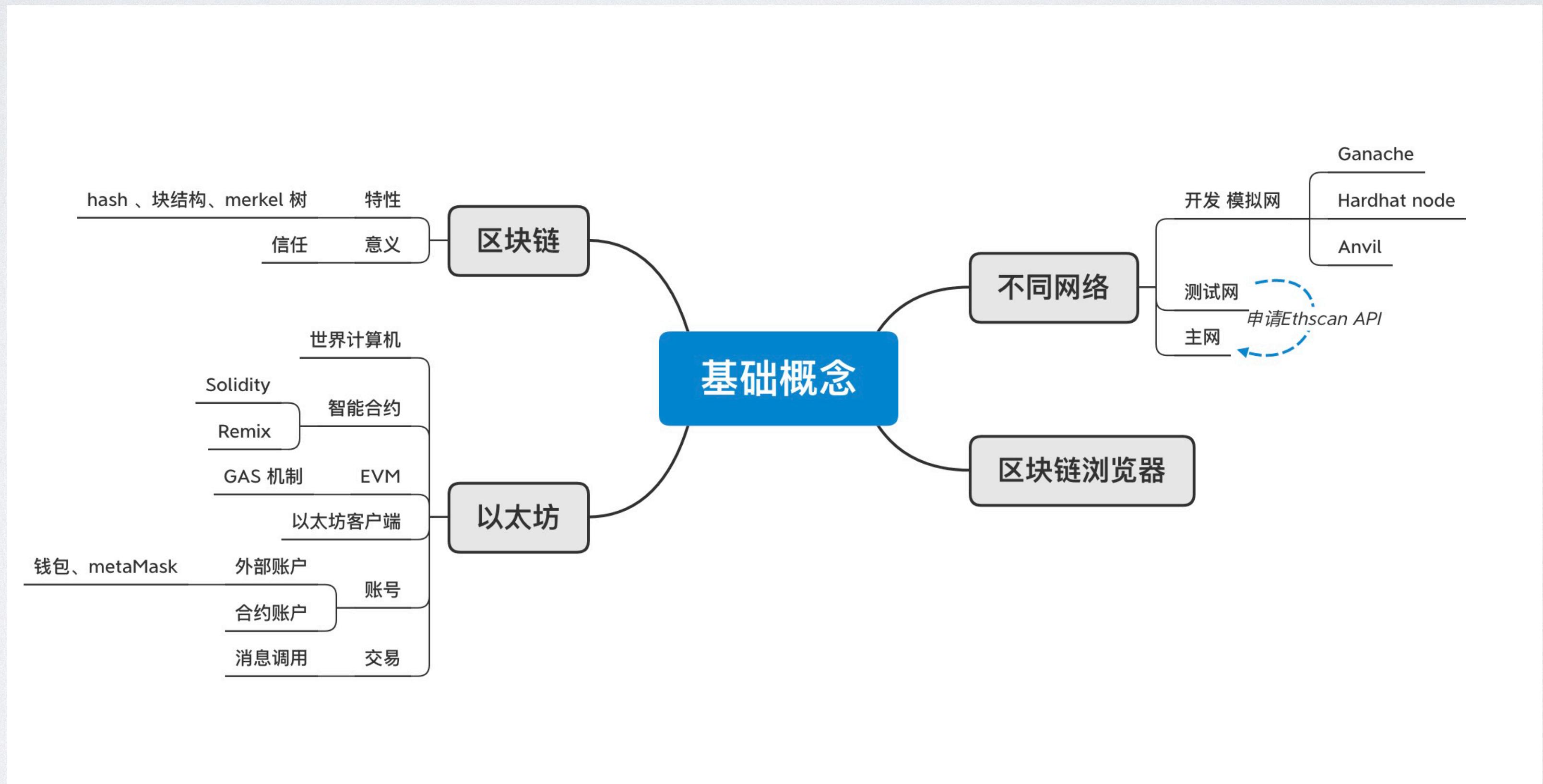
- Tiny 熊
- 创新工场（点心）、猎豹移动、合伙创业
- 登链社区发起人 (from 2017)
- 出版过几本区块链书籍：



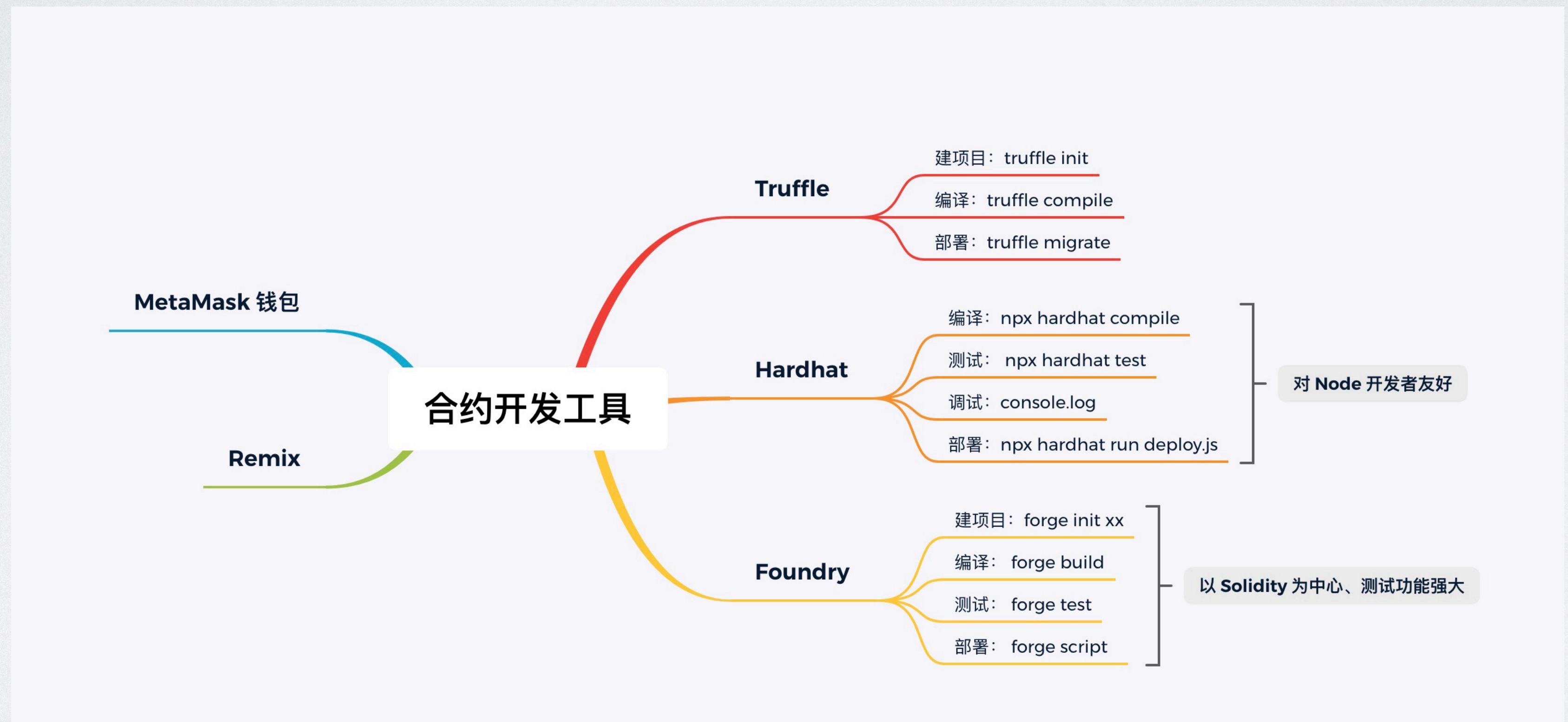
# 课程遇到的问题

- 课件在哪里
  - <https://learnblockchain.cn/course/28>
- 录播在哪里
  - <https://learnblockchain.cn/course/28>
- 代码在哪里
  - [https://github.com/xilibi2003/training\\_camp\\_2](https://github.com/xilibi2003/training_camp_2)
- 预习 + 复习：<https://decert.me/tutorial/solidity/>
- 技能 = 知识 + 实践
- 坚持 就是 胜利

# 复习



# 复习



# 习题解答

- 修改 Counter 合约，仅有部署者 可以调用 count();
- 使用 Hardhat 部署修改后的 Counter
- 使用 Hardhat 测试 Counter:
  - Case 1: 部署者成功调用 count()
  - Case 2: 其他地址调用 count() 失败
- 代码开源到区块浏览器 (npx hardhat verify ...) / 写上合约地址

代码库：[https://github.com/xilibi2003/training\\_camp\\_2](https://github.com/xilibi2003/training_camp_2)  
answer 分支

# 作业遇到问题

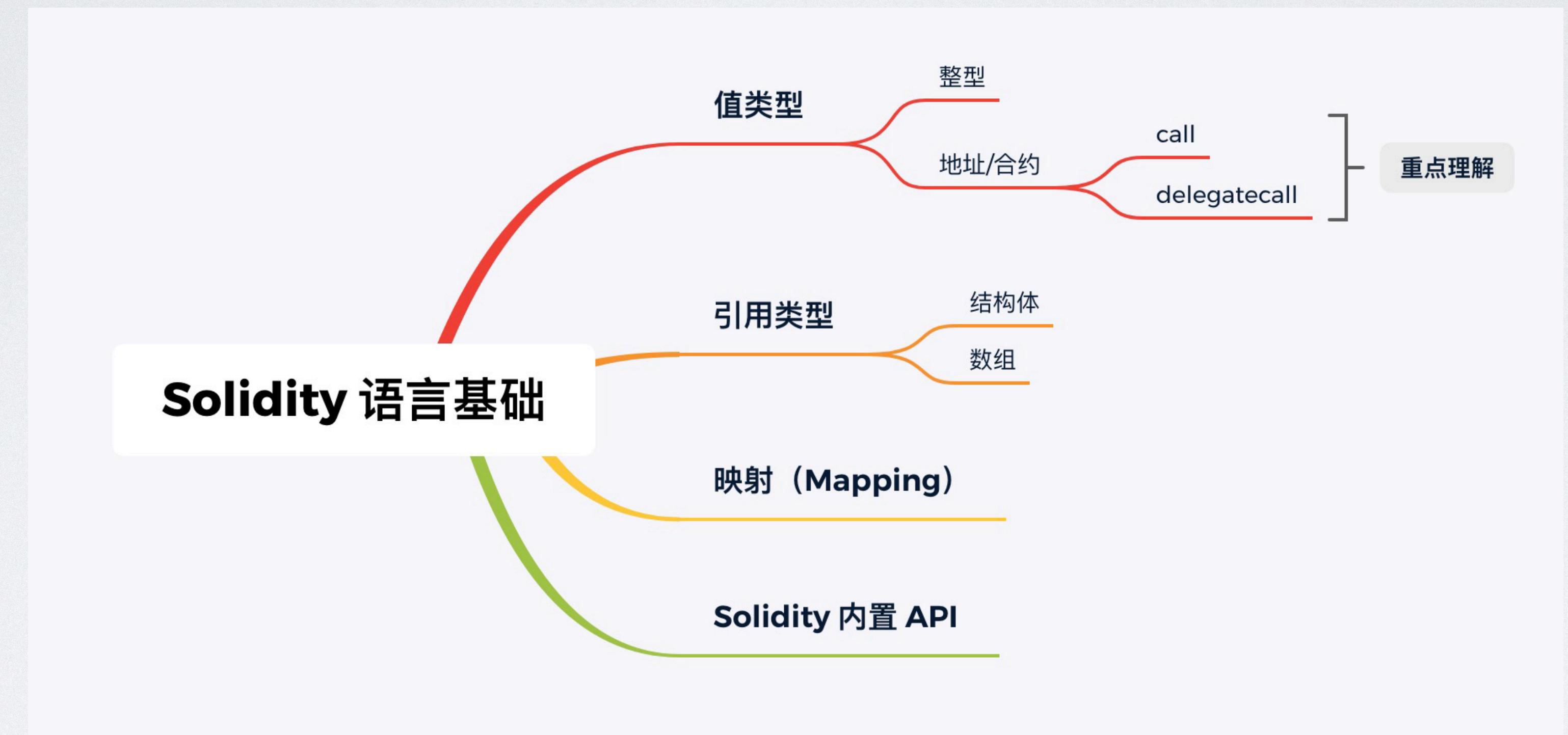
- 水龙头获取代币
- Etherscan API 申请（到对应主网）
- 项目配置项（hardhat.config.js）
- 网络问题（代码验证、编译器下载）
- 其他问题

代码库：[https://github.com/xilibi2003/training\\_camp\\_2](https://github.com/xilibi2003/training_camp_2)  
answer 分支

# SOLIDITY语言主要特性

- Solidity 基本类型、数组、结构体、映射
- Solidity API 介绍
- 合约函数、函数修改器、函数修饰符，及各类特殊函数
- 错误处理、合约继承、接口、库及 Openzeppelin 合约库
- 理解合约事件
- 理解ABI

# SOLIDITY 学习导图



# 合约的组成

```
pragma solidity ^0.8.0; 1. 编译器版本声明
contract Counter { 2. 定义合约
    uint public counter; 3. 状态变量
    constructor() {
        counter = 0;
    }
    function count() public { 4. 合约函数
        counter = counter + 1;
    }
}
```

# SOLIDITY 语言

- 静态类型、编译型、高级语言
- 针对 EVM 专门设计
- 受 C++、JavaScript 等语言影响
  - 如：变量声明、for 循环、重载函数的概念 来自于 C++
  - 函数关键字、导入语法来自于 JavaScript
- 文档
  - 中文：<https://learnblockchain.cn/docs/solidity/>
  - 英文：<https://docs.soliditylang.org/>

# SOLIDITY 数据类型

- Solidity 值类型
  - 布尔、**整型**、定长浮点型、定长字节数组、枚举、函数类型、地址类型
  - 十六进制常量、有理数和整型常量、字符串常量、地址常量
- Solidity 引用类型
  - **结构体**
  - **数组**
- 映射类型

# SOLIDITY – 整型

- 整型： int/uint , uint8 ... uint256

支持运算符

- 比较运算： `<=`, `<`, `==`, `!=`, `>=`, `>`
- 位运算： `&`, `|`, `^`(异或), `~`(位取反)
- 算术运算： `+`, `-`, `-(负)`, `*`, `/`, `%`(取余数), `**` (幂)
- 移位： `<<` (左移位), `>>`(右移位)

在使用整型时，要特别注意整型的大小及所能容纳的最大值和最小值，如uint8的最大值为0xff（255），最小值是0。从solidity 0.6.0 版本开始可以通过 `Type(T).min` 和 `Type(T).max` 获得整型的最小值与最大值。

# SOLIDITY – 整型

预测一下2个函数分别的结果是什么?  
为什么?

```
pragma solidity ^0.5.0;

contract testOverflow {
    function add1() public pure returns (uint8) {
        uint8 x = 128;
        uint8 y = x * 2;
        return y;
    }

    function add2() public pure returns (uint8) {
        uint8 i = 240;
        uint8 j = 16;
        uint8 k = i + j;
    }
}
```

# SOLIDITY – 地址类型

- Solidity 使用地址类型来表示一个账号，地址类型有两种形式
  - address: 一个20字节的值。
  - address payable: 表示可支付地址，与address相同也是20字节，不过它有成员函数transfer和send。
- 成员函数
  - <address>.balance(uint256): 返回地址的余额
  - <address payable>.transfer(uint256 amount): 向地址发送以太币，失败时抛出异常 (gas: 2300)
  - <address payable>.send(uint256 amount) returns (bool): 向地址发送以太币，失败时返回false

# SOLIDITY – 地址类型

```
pragma solidity ^0.6.0;

contract testAddr {

    function testTrasfer(address payable x) public {
        address myAddress = address(this); 合约转换为地址类型

        if (myAddress.balance >= 10) {
            x.transfer(10);
        }
    }
}
```

给一个合约地址转账，即上面代码 x 是合约地址时，合约的receive函数或fallback函数会随着transfer调用一起执行

# SOLIDITY – 地址类型

- 3个底层成员函数: `call`, `delegatecall`, `staticcall`
  - `<address>.call(bytes memory) returns (bool, bytes memory)`

通常用于合约交互，直接控制编码的方式调用合约函数。

例如调用合约的`register(string)`方法：

```
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
```

# SOLIDITY – 地址高级用法

- **call()**

切换上下文，附加gas {gas: }，附加value {value: }

addr.call{value: 1 ether}("") 功能等价 transfer(1 ether)，但是没有 gas 限制

失败不是发生异常，一定要检查返回值

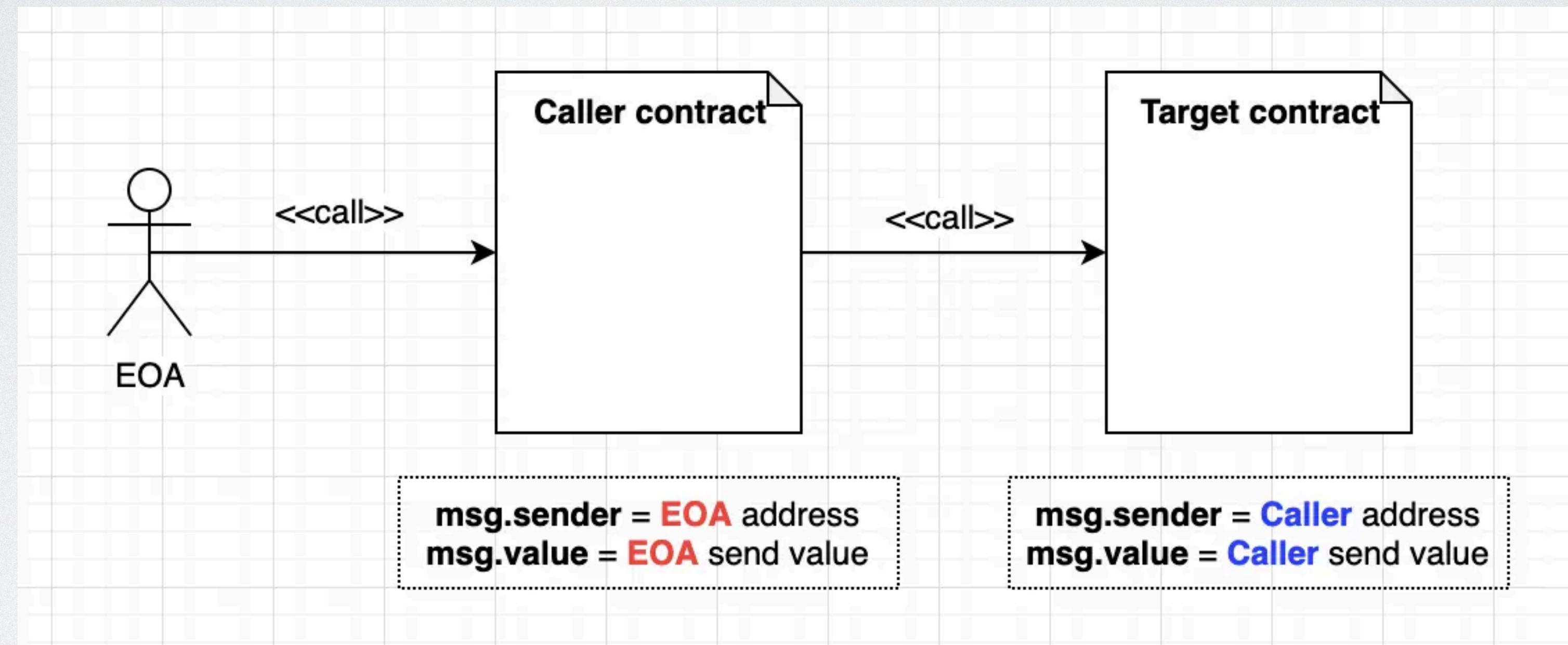
- **delegatecall()**

保持上下文

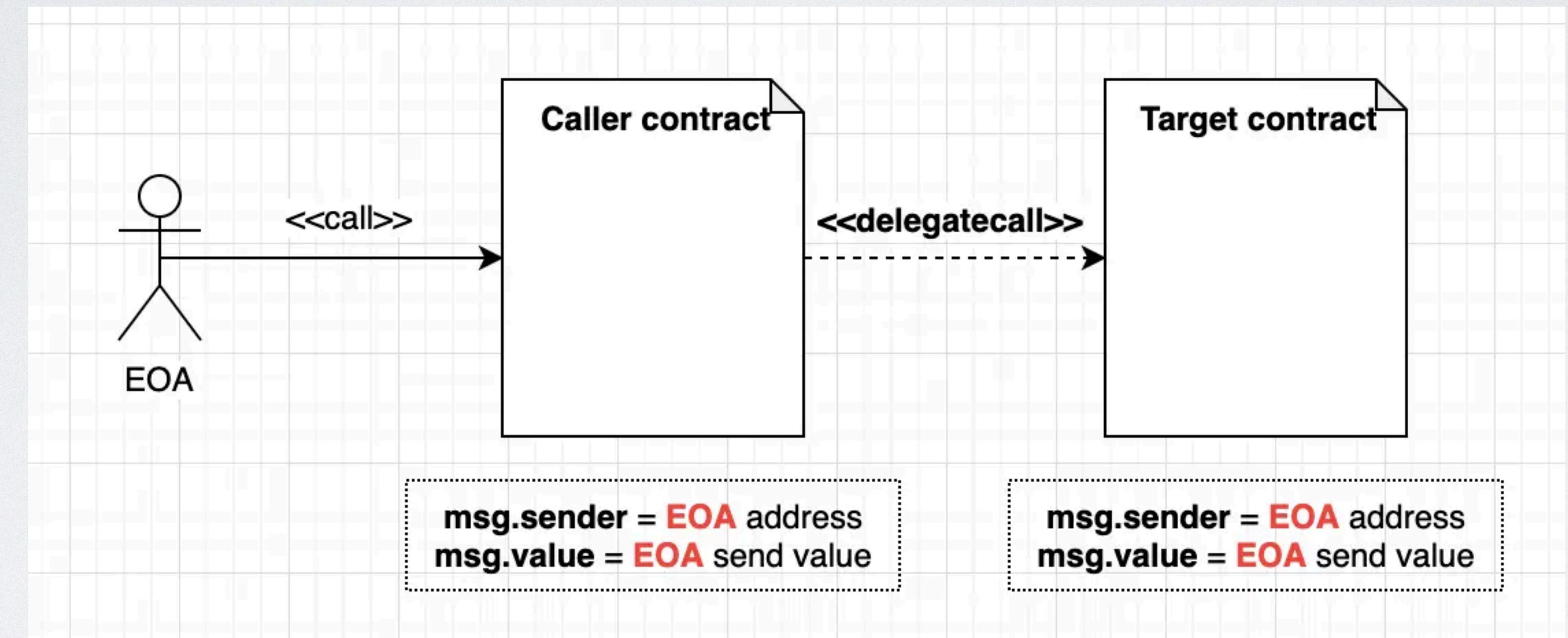
不支持附加value {value: }

[https://github.com/Uniswap/solidity-lib/blob/master/contracts/libraries/  
TransferHelpers.sol](https://github.com/Uniswap/solidity-lib/blob/master/contracts/libraries/TransferHelpers.sol)

# SOLIDITY – 地址 – CALL



# SOLIDITY – 地址 – DELEGATECALL



# SOLIDITY – 合约类型

```
pragma solidity ^0.6.0;

contract Hello {
    function sayHi() public {
    }
}
```

- 每个合约都是一个类型，可声明一个合约类型。  
如：Hello c; 则可以使用c.sayHi() 调用函数
- 合约可以显式转换为address类型，从而可以使用地址类型的成员函数。

# SOLIDITY – 引用类型

- 值类型赋值时总是完整拷贝。而复杂类型占用的空间较大 ( $>32$ 个字节) , 拷贝开销很大, 这时就可以使用引用的方式, 即通过多个不同名称的变量指向一个值。
- 引用类型都有一个额外属性来标识数据的存储位置:
  - memory (内存) : 生命周期只存在于函数调用期间
  - storage (存储) : 状态变量保存的位置, gas开销最大
  - calldata (调用数据) : 用于函数参数不可变存储区域

# SOLIDITY – 数组

- $T[k]$  : 元素类型为T， 固定长度为k的数组
- $T[]$  : 元素类型为T， 长度动态调整
- 数组通过下标进行访问， 序号是从0开始
- bytes、 string 是一种特殊的数组
  - bytes是动态分配大小字节的数组， 类似于byte[], 但是bytes的gas费用更低， bytes和string都可以用来表达字符串， 对任意长度的原始字节数据使用bytes， 对任意长度字符串（UTF-8）数据使用string。

# SOLIDITY – 数组

- 成员
  - Length属性：表示当前数组的长度
  - push(): 添加新的零初始化元素到数组末尾，返回引用
  - push(x): 数组末尾添加一个给定的元素
  - pop(): 从数组末尾删除元素

# SOLIDITY – 数组

```
pragma solidity ^0.8.0;

contract testArray {
    uint[10] tens;
    uint[] public numbers;

    function test(uint len) public pure {
        string[4] memory adaArr = ["This", "is", "an", "array"];
        uint[] memory c = new uint[](len);
    }

    function add(uint x) public {
        numbers.push(x);
    }
}
```

# SOLIDITY – 数组

- gas 问题
- 移除元素
- 遍历操作

```
function dosome() public {
    uint len = numbers.length;
    for (uint i = 0; i < len; i++) {
        // do...
    }
}

function remove(uint index) public {
    uint len = numbers.length;
    if (i == len - 1) {
        numbers.pop();
        break;
    } else {
        numbers[index] = numbers[len - 1];
        epoches.pop();
        break;
    }
}
```

# SOLIDITY – 结构体

使用Struct 声明一个结构体， 定义一个新类型。

```
contract testStruct {
    struct Funder {
        address addr;
        uint amount;
    }

    mapping (uint => Funder) funders;
    function contribute(uint id) public payable {
        funders[id] = Funder({addr: msg.sender, amount: msg.value});
        funders[id] = Funder(msg.sender, msg.value);
    }
    function getFund(uint id) public view returns (address, uint ) {
        return (funders[id].addr, funders[id].amount);
    }
}
```

# SOLIDITY – 映射

- 声明形式： mapping(KeyType => ValueType) , 例如： mapping(address => uint) public balances;
- 使用方式类似数组，通过 key 访问，例如： balances[userAddr];
- 映射没有长度、没有 key 的集合或 value 的集合的概念
- 只能作为状态变量
- 如果访问一个不存在的键，返回的是默认值。

# SOLIDITY – 映射

```
pragma solidity >=0.8.0;

contract MappingExample {
    mapping(address => uint) public balances;

    function update(uint newBalance) public {
        balances[msg.sender] = newBalance;
    }
}
```

# SOLIDITY – 全局变量及函数

- 区块和交易
- ABI 编码
- 错误处理
- 数学及加密
- 地址及合约

<https://learnblockchain.cn/docs/solidity/units-and-global-variables.html#special-variables-and-functions>

# SOLIDITY – 全局变量及函数

- `block.number` ( `uint` ): 当前区块号
- `block.timestamp` ( `uint` ): 自 unix epoch 起始当前区块以秒计的时间戳
- `msg.sender` ( `address` ): 消息发送者 (当前调用)
- `msg.value` ( `uint` ): 随消息发送的 wei 的数量
- `tx.origin` ( `address payable` ): 交易发起者 (完全的调用链)
- ...

<https://learnblockchain.cn/docs/solidity/units-and-global-variables.html#special-variables-and-functions>

# Q & A

# 练习题

- 编写一个Bank合约：
- 通过 Metamask 向Bank合约转账ETH
- 在Bank合约记录每个地址转账金额
- 编写 Bank合约withdraw(), 实现提取出所有的 ETH

# SOLIDITY – 合约

- 使用contract关键字来声明一个合约，一个合约通常由状态变量、函数、函数修改器以及事件组成。

```
pragma solidity >=0.8.0;

contract C {
    function f(uint a) private pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

# SOLIDITY – 合约

- 创建合约的几个方法：
  - 外部部署 (Remix/Hardhat/Truffle) Web.js
  - 合约使用New
  - 最小代理合约 (克隆)：
    - <https://eips.ethereum.org/EIPS/eip-1167>
    - <https://github.com/optionality/clone-factory>
  - Create2
    - C c = `new` C{salt: \_salt}();

# SOLIDITY – 合约地址

- 合约地址的确认
  - 根据创建者 (sender) 的地址以及创建者发送过的交易数量 (nonce) 来计算确定
    - keccak256(rlp.encode([normalize\_address(sender), nonce]))[12:]
  - Create2
    - keccak256(0xff ++ senderAddress ++ salt ++ keccak256(init\_code))[12:]

# SOLIDITY – 合约

内外

外部访问

内部

内部及继承

- 使用public、private、external、internal 可见性关键字来控制变量和函数是否被外部使用

```
pragma solidity >=0.8.0;

contract C {
    function f(uint a) public pure returns (uint b) { return a + 1; }
    function setData(uint a) internal { data = a; }
    uint public data;
}
```

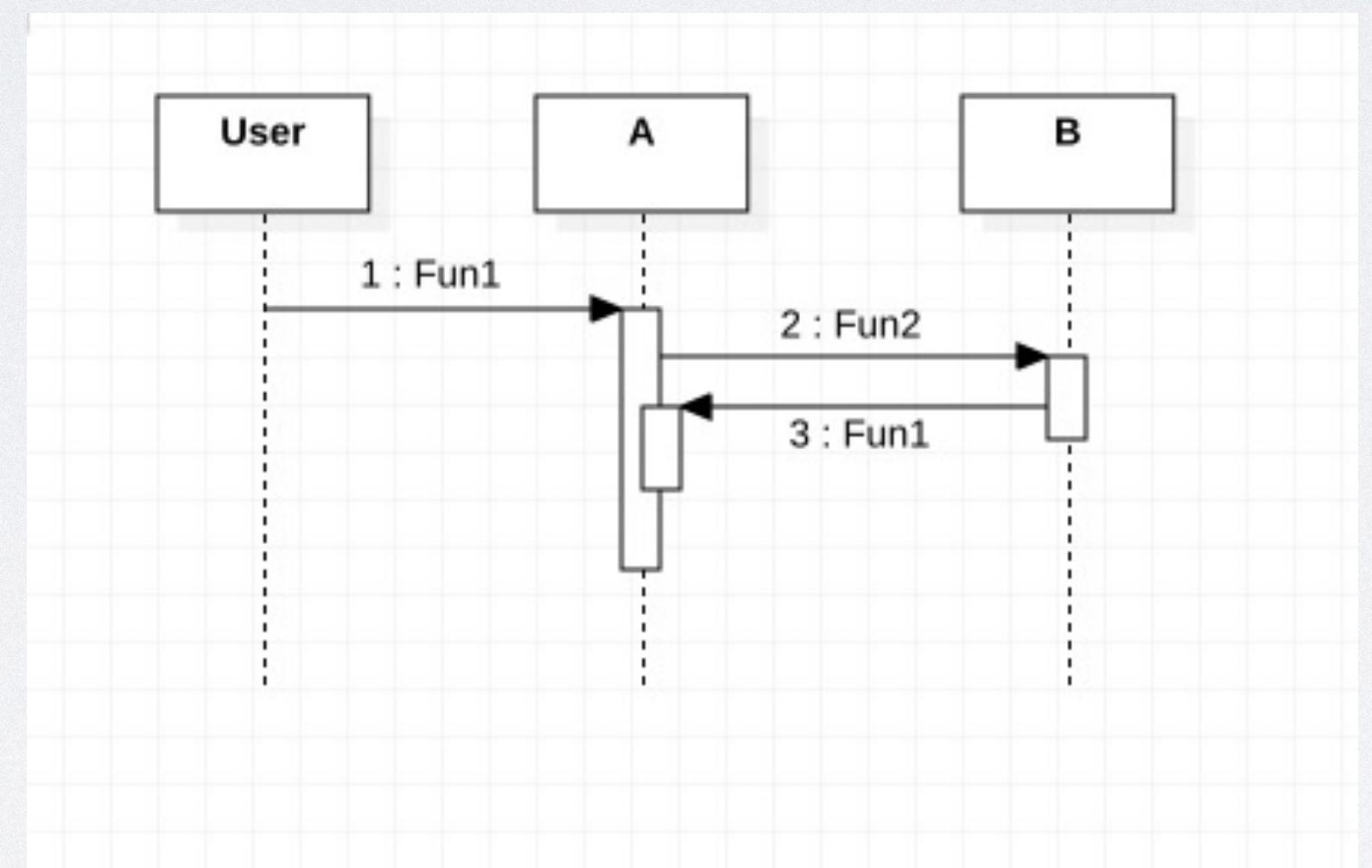
# SOLIDITY – 合约函数

- 构造函数(constructor): 初始化逻辑
- 视图函数(view)、纯函数:(pure) 不修改状态，不支付手续费
- getter 函数: 所有 public 状态变量创建 getter 函数
- Payable 修饰符: 表示一个函数可以接收以太币
- receive 函数: 接收以太币时回调。
- fallback 函数: 没有匹配函数标识符时, fallback 会被调用, 如果是转账时, 没有receive也有调用fallback
- 函数修改器 (modifier) : 可用来改变一个函数的行为, 如检查输入条件、控制访问、重入控制

# SOLIDITY – 重入问题

- 调用外部函数时，要时刻注意重入问题：

- 重入



## 第2周

```
pragma solidity ^0.8.0;
contract testFunc {
    address public owner;
    uint private deposited;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;          函数修改器修饰函数时，函数体被插入到“_”
    }
    fallback() external payable {
        deposited += msg.value;
    }
    function getDeposited() public view returns(uint) {
        return deposited;
    }
    function withdraw() public onlyOwner {
        payable(owner).transfer(deposited);
    }
}
```

# SOLIDITY – 错误处理

- 在程序发生错误时的处理方式：EVM通过回退状态来处理错误的，以便保证状态修改的**事务性**
- assert()和require()用来进行条件检查，并在条件不满足时抛出异常
- revert(): 终止运行并撤销状态更改
- try/catch: 捕获合约中外部调用的异常,
  - 注意：out of gas 错误不是程序异常，错误不能捕获。

## 第2周

```
pragma solidity ^0.8.0;

contract Foo {
    function myFunc(uint x) public pure returns (uint ) {
        require(x != 0, "require failed");
        return x + 1;
    }
}

contract trycatch {

    Foo public foo;
    uint public y;
    constructor() {
        foo = new Foo();
    }

    function tryCatchExternalCall(uint _i) public {
        try foo.myFunc(_i) returns (uint result) {
            y = result;
        } catch {
        }
    }
}
```

# SOLIDITY – 继承

- 和大多数高级语言一样，Solidity 也支持继承
- 使用关键字 `is`
- 继承时，链上实际只有一个合约被创建，基类合约的代码会被编译进派生合约。
- 派生合约可以访问基类合约内的所有非私有（`private`）成员，因此内部（`internal`）函数和状态变量在派生合约里是可以直接使用的

## 第2周

```
pragma solidity ^0.8.0;

contract A {
    uint public a;
    constructor() {
        a = 1;
    }
}

contract B is A {
    uint public b ;
    constructor() {
        b = 2;
    }
}
```

在部署B时候，可以查看到a为1， b为2。

# SOLIDITY – 继承、抽象合约

- abstract 抽象合约
  - 不能被部署，可包含没有实现的纯虚函数
- super：调用父合约函数
- virtual：表示函数可以被重写
- override：表示重写了父合约函数

# SOLIDITY – 接口

- 函数的抽象，作为一个类型声明，广泛用于合约之间的调用
- 不可以有实现的函数
- 不能继承自其他接口
- 没有构造方法
- 没有状态变量

## 第2周

```
pragma solidity ^0.8.0;

contract Counter {
    uint public count;

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

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

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

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

# SOLIDITY – 库

- 与合约类似（一个特殊合约），是函数的封装，用于代码复用。
- 如果库函数都是 `internal` 的，库代码会嵌入到合约。
- 如果库函数有`external`或 `public`，库需要单独部署，并在部署合约时进行链接，使用委托调用
- 没有状态变量
- 不能给库发送 Ether
- 给类型扩展功能：Using lib for type; 如：`using SafeMath for uint;`

```
pragma solidity ^0.8.0;

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

        return z;
    }
}

contract TestLib {
    using SafeMath for uint;

    function testAdd(uint x, uint y) public pure returns (uint) {
        return x.add(y);
    }
}
```

# SOLIDITY – 链接外部库

```
const ExLib = await hre.ethers.getContractFactory("Library");
const lib = await ExLib.deploy();
await lib.deployed();

await hre.ethers.getContractFactory("TestExLib", {
  libraries: {
    Library: lib.address,
  },
});
```

# SOLIDITY – OpenZeppelin

- 善于复用库，不仅提高效率，还可以提高安全性
- OpenZeppelin 功能丰富、经过反复验证的库函数集合。

下一周进一步介绍OpenZeppelin

# SOLIDITY – 事件

- 合约与外部世界的重要接口，通知外部世界链上状态的变化
- 事件有时也作为便宜的存储
- 使用关键字 event 定义事件，事件不需要实现
- 使用关键字 emit 触发事件
- 事件中使用indexed修饰，表示对这个字段建立索引，方便外部对该字段过滤查找

## 第2周

```
pragma solidity ^0.8.0;

contract testEvent {
    constructor() {
    }

    event Deposit(address indexed _from, uint _value);

    function deposit(uint value) public {
        emit Deposit(msg.sender, value);
    }
}
```

事件将记录在日志之中，下周将介绍如何从交易收据中解析日志

# ABI

- ABI: Application Binary Interface 应用程序二进制接口
- ABI 接口描述: 定义如何与合约交互
- ABI 编码
  - 函数选择器: 对函数签名计算Keccak-256哈希, 取前 4 个字节
  - 参数编码

# ABI

调用一个合约函数 = 向合约地址发送一个交易

交易的内容就是 ABI 编  
码数据

# Calling a Smart Contract



Address: 0x0123456.....

## 1 Convert function call to HEX

myFunction(parameters) → → 0abcdef0123456789.....

## 2 Put the Information into Transaction object

```
{  
  "to": "0x0123456.....",  
  "value": 0, // No need to send money here  
  "data": "0abcdef0123456789....."  
}
```

## 3 Sign the Transaction with your Private key

{ ... } + Private Key → → 0xfedcba9876...

Signed Transaction  
Can only be decrypted with YOUR public key  
Only you can have sent this transaction

## 4 Send the transaction to the Ethereum Network



第2周

# ABI

bytes4(keccak256("count()")) = 0x06661abd

Tx Hash: <https://ropsten.etherscan.io/tx/0xaafc79373cb38081743fe5f0ba745c6846c6b08f375fd028556b4e52330088b>

<https://www.4byte.directory/>

The screenshot shows the ABI details for the function `count()`. It includes fields for Gas Price (0.00000002 Ether), Gas Limit & Usage by Txn (41,614 / 41,614 (100%)), and Others (Nonce: 1, Position: 1). A box highlights the MethodID: `0x06661abd`. A button labeled "View Input As" is also visible.

```
(bool success,) = address(x).call("0x06661abd");
require(success);
```

# 推荐SOLIDITY学习资料

- <https://ethernaut.openzeppelin.com/>
- <https://cryptozombies.io/en/course/>
- <https://solidity-by-example.org/>
- <https://learnblockchain.cn/column/1>

# Q & A

# 练习题

- 编写合约Score，用于记录学生（地址）分数：
  - 仅有老师（用modifier权限控制）可以添加和修改学生分数
  - 分数不可以大于 100；
- 编写合约Teacher 作为老师，通过 IScore 接口调用修改学生分数。