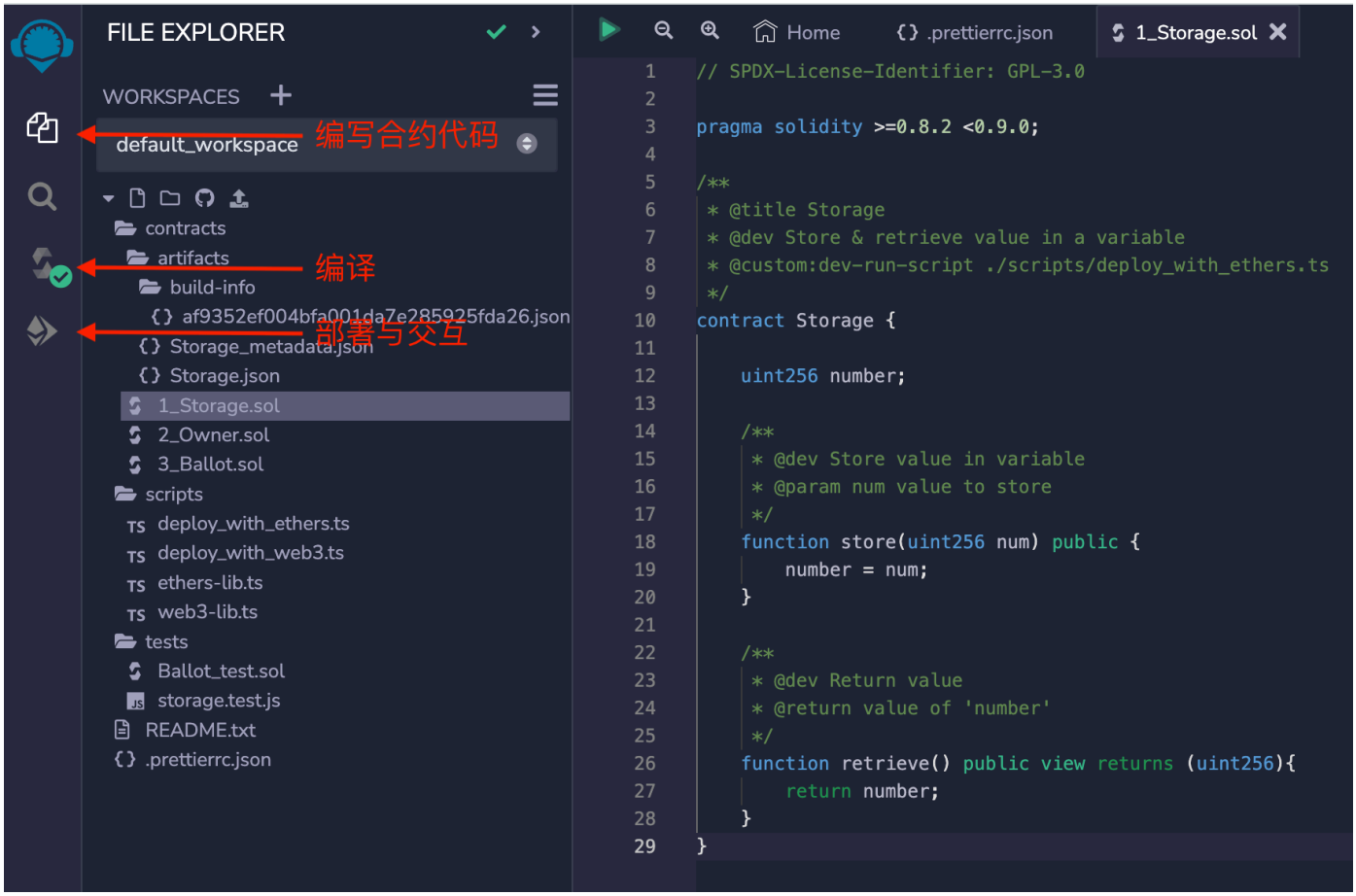
以太坊开发入门实战 第三课 作业点评

课程回顾

Remix的使用

Remix是以太坊官方开源的Solidity在线集成开发环境。

可以在网页内完成智能合约的开发、编译、部署、调试与交互。



Solidity语言基础

值类型

- 布尔类型 `bool` true/false
- 整型 `int` / `uint` (int256/uint256) 8~256位，以8为步长
- 定长浮点型 Solidity没有完全支持该类型，所以合约中一般只用整型
- 地址类型 20字节，账户地址大小 => 地址类型(`address`)成员变量
- 定长字节数组 `bytes1` , `bytes2` ...
- 变长字节数组 `bytes` , `string`

- 枚举类型 `enum ActionChoices { Goleft, GoRight, GoStraight, SitStill}`
- 函数类型

  `function(<parameter types>)[private|internal|external|public][pure|view|payable] [returns (return types)]`

  在Solidity中，函数的可见性（visibility）可以用`private`、`internal`、`external`或`public`关键字来标识，这些关键字用于决定函数可以被哪些合约调用。

  - `private`：该函数只能在当前合约内部访问，子合约无法调用父合约的`private`函数。
  - `internal`：该函数只能在当前合约内部以及继承自当前合约的合约中访问，不能从其他合约中访问。
  - `external`：该函数只能从当前合约外部调用，即只能通过交易来调用该函数，不能在当前合约内部调用。在外部调用时需要使用合约地址来调用该函数。
  - `public`：该函数既可以在当前合约内部调用，也可以在外部通过合约地址进行调用。当函数被声明为`public`时，编译器会为该函数自动生成一个访问器（`getter`）函数，用于返回公共变量的值。

  在Solidity中，除了可见性关键字外，还有一些函数修饰符可以用来指定函数的属性。其中最常用的包括`pure`、`view`和`payable`。

  - `pure`：该函数不会读取或修改合约状态，也不会调用其他合约的函数，仅仅依赖于输入参数并返回一个值。这使得它可以被安全地调用而无需支付gas费用，因为不会有状态变化和交易费用。
  - `view`：该函数不会修改合约状态，但是它可以读取合约的状态和存储，它是一个只读函数。调用一个`view`函数是不需要花费任何gas费用的，因为不会有状态变化和交易费用。
  - `payable`：该函数可以接收以太币，并会修改合约状态。当一个函数被标记为`payable`时，就可以在调用它的时候向它发送以太币。调用一个标记为`payable`的函数时，必须要支付一定的gas费用来完成交易。

地址类型

- `balance`：作为属性，直接查询该地址下的以太币余额
  - 举例

```
contract MyContract {
    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}

```

- `transfer`：作为方法，向该地址发送以太币(Ether)

- `transfer` 函数可以在任何函数或合约中使用，但只能向外部地址或合约地址发送以太币，不能发送到另一个合约的函数。
- 需要注意的是，接收方地址必须是一个 `payable` 地址，这意味着它必须是一个可以接收以太币的合约或外部帐户。
- 举例

```
contract MyContract {
    function transferEther(address payable recipient, uint256 amount) public {
        require(address(this).balance >= amount, "Insufficient balance.");
        recipient.transfer(amount);
    }
}

```

在这个示例中，`transferEther()` 函数接受两个参数：接收方地址 `recipient` 和要转移的以太币数量 `amount`。函数首先检查当前合约的以太币余额是否足够支付转移金额，如果余额不足，将会抛出一个异常。如果余额足够，函数将使用 `transfer` 关键字将指定数量的以太币转移到指定地址。

`call`、`delegatecall`、`staticcall`

在 Solidity 中，可以使用三种不同的方法来调用其他合约或外部地址的函数：`call`、`delegatecall` 和 `staticcall`。这三种方法具有不同的用途和行为，下面分别介绍它们的用法。

- **`call`**

`call` 方法是 Solidity 中最基本的调用方法。它允许你调用其他合约或外部地址的函数，并将数据传递给它们。`call` 方法返回两个值：一个布尔值，表示调用是否成功，以及一个返回值（如果有的话）。

以下是一个使用 `call` 调用外部合约函数的示例：

```
contract Caller {
    function callAnotherContract(address contractAddress, uint256 inputValue) public returns (uint256) {
        (bool success, bytes memory result) = contractAddress.call(abi.encodeWithSignature("foo(uint256)", inputValue));
        require(success, "External call failed.");
        return abi.decode(result, (uint256));
    }
}
```

在这个示例中，`callAnotherContract` 函数调用了名为 `foo` 的函数，并将一个 `uint256` 类型的参数传递给它。如果调用成功，函数将返回一个 `uint256` 类型的结果。

→补充

`abi.encodeWithSignature`：用于将函数名和参数编码为一个字节数组，以便在调用合约函数时使用。它的用法示例如下：

```
bytes memory data = abi.encodeWithSignature("myFunction(uint256,address)", 123, 0x1234567890123456789012345678901234567890);
```

即参数是一个函数签名，以及用逗号隔开的函数实参。

编码后的数据可以用于调用合约函数，例如：

```
(bool success, bytes memory result) = contractAddress.call(data);
```

在这个例子中，我们使用 `contractAddress.call` 函数调用合约函数，并传递编码后的函数名和参数。`call` 函数将执行合约中与传递的数据匹配的函数，并返回一个布尔值表示是否成功以及一个包含返回值的字节数组。

- **`delegatecall`**

`delegatecall` 方法与 `call` 方法类似，但它在执行时会将当前合约的上下文传递给被调用的合约。这意味着被调用的合约可以访问当前合约的状态变量和存储，但调用合约不能访问被调用合约的状态。

即`delegatecall`在自身合约环境执行外部合约逻辑，call是在外部合约环境执行逻辑。

- **`staticcall`**

`staticcall` 方法与 `call` 方法类似，但它不能修改合约的状态。这意味着被调用的合约不能修改任何状态变量或存储。当调用的外部合约函数修改了状态，就会回滚失败。

`mapping`

`mapping` 是一种用于存储键值对的数据结构。它类似于哈希表或字典，可以通过键访问值。与数组不同，`mapping` 不需要预先定义其大小，并且可以根据需要动态添加和删除元素。

例如，以下代码声明了一个将 `address` 映射到 `uint256` 的 `mapping` 变量`balances`：

```
mapping(address => uint256) balances;
```

可以通过以下方式将值存储到 `balances` 中：

```
balances[msg.sender] = 100;
```

需要注意的是，`mapping` 只能在存储变量中使用，不能作为函数的返回值或参数。`mapping`不存储key，因此没有长度以及key或value的集合概念。`mapping` 也不能被复制或迭代。

其他

- 数组：bytes string 是特殊的数组类型；方法有length、push(x)、pop()等
- 结构体：可以作为元素用在数组或mapping中
- 数据位置：关键字 `storage` | `memory`
  - 函数参数和返回值默认是`memory`，局部数组或结构体变量默认是`storage`，合约状态变量强制是`storage`；
  - `storage`与`memory`之间两两赋值，`storage`向状态变量赋值创建独立拷贝；
  - 状态变量向`storage`赋值传递引用；
  - `memory`向`memory`赋值不会创建拷贝；
  - 只有状态变量可以用`mapping`类型。

单位

- 以太币单位
  - 1 wei == 1
  - 1 gwei == 1e9
  - 1 ether == 1e18
- 时间单位
  - 秒是默认时间单位
  - 1 == 1 second

全局变量

详见文档

- 区块与交易相关:
  - blockhash(uint blockNumber) returns(bytes32) 最近前256区块，超出默认返回0
  - block.number　　区块高度
  - block.timestamp　　当前区块出块时间戳
  - msg.data　　上链交易的data
  - msg.value　　交易发送的以太数量
  - msg.sender　　当前调用的账户地址(合约or外部账户)
  - tx.origin　　交易的发起账户地址

注：对于每一个外部合约调用，`msg.sender`, `msg.value`, `msg.data` 都会变化

错误处理

- assert(bool condition)
- require(bool condition)、require(bool condition, string memory message)
- revert()、revert(string memory reason)

合约

- `getter`函数：为public状态变量自动创建。内部访问被认为是状态变量，外部访问被认为是函数；
- 函数修饰器 modifier：在函数执行之前自动检查某个条件；
- constant && immutable 变量：constant 编译时确定，immutable 合约创建初始化确定；
- 特殊函数
  - constructor() public
  - receive() external payable {}
  - fallback() external
- 继承、抽象合约、接口
- Library：若通过delegatecall 调用，只需部署一次，无状态。若直接调用，函数为view or pure。

作业点评

示例代码：

```
//创建不同的募资活动，用来募集以太坊
//记录相应活动下的募资总体信息（参与人数，募集的以太坊数量），以及记录参与的用户地址及投入数量
//业务逻辑（用户参与，添加新的募集活动，活动结束后进行资金领取）

pragma solidity 0.8.11;

contract CrowdFundingStorage {
    struct Campaign{
         address payable receiver;  //募资接收地址
         uint numFunders;   //募捐人数
         uint fundingGoal;  //募捐目标金额
         uint totalAmount;  //募捐实际总金额
     }

    struct Funder{
        address addr;  //募捐人地址
        uint amount;   //募捐人募捐总金额
    }

    //活动数量
    uint public numCampaigns;
```

```
    //k 募资活动编号，v 募捐活动
    mapping(uint => Campaign) campaigns;
    //k 募资活动编号，v 募捐人列表
    mapping(uint => Funder[]) funders;
    //k1 募资活动编号，k2 参与人地址，v false 未参与，true 已参与
    mapping(uint => mapping(address => bool)) public isParticipate;
}

//支持多继承
contract ComdFunding is CrowdFundingStorage{
    address immutable onwer;
    constructor(){
        //谁发布合约谁就是onwer
        onwer = msg.sender;
    }

    //判定用户不能参与相同活动
    modifier judgeParticipate(uint compaignID) {
        require(isParticipate[compaignID][msg.sender] == false);
        _; //通过require后，继续执行函数内容
    }

    //只有合约创建人才能创建募资活动
    modifier isOwner() {
        require(msg.sender == onwer);
        _;
    }

    //创建募资活动(入参：receiver 募捐活动接收地址，goal 募捐目标金额；返回参数：募捐活动编号)
    function newCampaign(address payable receiver, uint goal) external isOwner()
returns(uint compaignID) {
       compaignID = numCampaigns++;     //在returns()中声明函数内部变量
       Campaign storage c = campaigns[compaignID];    //见[防止越界——Mapping]中的解释
       c.receiver = receiver;
       c.fundingGoal = goal;
    }

    //参与募资活动
    function bid(uint campaignID) external payable judgeParticipate(campaignID){
        Campaign storage c = campaigns[campaignID];
        c.totalAmount += msg.value;
        c.numFunders += 1;

        funders[campaignID].push(
            Funder({
                addr: msg.sender,
                amount: msg.value
            })
        );
```

```
70
71          isParticipate[campaignID][msg.sender] = true;
72      }
73
74      //活动结束后领取资金
75      function withdraw(uint campaignID) external returns(bool reached) {
76          Campaign storage c = campaigns[campaignID];
77          if (c.fundingGoal > c.totalAmount) {
78              return false;
79          }
80          uint amount = c.totalAmount;
81          c.totalAmount = 0;
82          c.receiver.transfer(amount);
83          return true;
84      }
85  }
```

知识点补充

在returns()中声明函数内部变量(命名式返回)

在 Solidity 中，在 `returns()` 关键字中声明函数内部变量是完全合法的。这个变量可以用于在函数执行过程中进行计算，并在函数返回时一起返回。

即可以在**returns中表明返回变量的名称**，solidity就会**自动**给这些变量**初始化**，并**自动返回**这些函数的值，**不需要加return**。

以下是一个示例代码：

```
1  solidityCopy code
2  function calculate(uint256 x, uint256 y) public pure returns (uint256 result,
   uint256 remainder) {
3      result = x / y;
4      remainder = x % y;
5  }
```

在这个例子中，我们声明了两个变量 `result` 和 `remainder`，并在函数内部进行计算。在 `returns()` 关键字中，我们可以指定这两个变量作为返回值。

防止越界

Mapping

当尝试使用一个在 `mapping` 中不存在的键（即一个不存在的 key）时，Solidity不会引发异常或错误，而是返回该键的默认值。

对于 `mapping` 类型，不存在的键的默认值是与映射值类型匹配的初始值。例如，如果声明了一个 `mapping(address => uint)` 类型的映射，并尝试使用一个不存在的地址去访问映射，则会返回默认值 `0`。

所以上面代码中的`Campaign storage c = campaigns[compaignID];`相当于在mapping中的compaignID索引下创建了个`Campaign`类型的空结构体，并将引用传递给`c`（初始化`c`），并在下面对`c`进行修改，修改的同时mapping中的compaignID索引下的结构体也会发生改变。

Array

如果尝试访问数组的不存在的元素，就会发生数组越界错误，这可能会导致运行时错误。

具体来说，如果你尝试访问一个大于等于数组长度的下标，将会触发一个“索引越界”的异常，导致函数的执行被中断，并且所有的状态修改都会被撤销，所有已使用的 gas 也将被返还。（*在 Solidity 中，如果在调用函数时发生异常，所有使用的 gas 都将被回退。这是因为在 EVM 中，如果发生异常，虚拟机将恢复到执行函数之前的状态，并且当前交易将被标记为失败。因此，所有使用的 gas 都将被返还给调用者。*）因此，在 Solidity 中需要格外注意数组下标的范围，以避免出现这种情况。

为了避免数组越界错误，在处理数组之前，应始终检查数组的长度，确保不会尝试访问超出数组长度范围的元素。在 Solidity 0.8.0 及更高版本中，可以使用 `Array` 类型来避免数组越界错误。`Array` 类型自动管理数组的长度，并提供了一些有用的方法来操作数组。

下面是一个防止数组越界的示例代码，使用 `require` 语句检查要访问的元素是否在数组的范围内：

```
pragma solidity ^0.8.0;

contract SafeArray {
    uint[] public myArray; //声明一个初始长度为0的Array

    function addValue(uint _value) public {
        myArray.push(_value);
    }

    function getValue(uint _index) public view returns (uint) {
        require(_index < myArray.length, "Index out of range"); //检查是否越界
        return myArray[_index];
    }

    function getLength() public view returns (uint) {
        return myArray.length;
    }
}

```

在上面的示例中，`getValue` 函数在访问数组元素之前，使用 `require` 语句检查要访问的索引 `_index` 是否小于数组长度 `myArray.length`。如果 `_index` 大于或等于 `myArray.length`，就会触发一个错误，并显示错误消息 "Index out of range"，防止数组越界。

使用这种方式，可以保证在访问数组时不会出现越界错误。

Gas优化技巧

当我们在编写 Solidity 智能合约时，我们需要考虑优化 gas 消耗。Gas 是以太坊中的计价单位，它表示了执行一项操作所需的计算资源。我们需要尽可能减少智能合约的 gas 消耗，以节省成本。

以下是一些常用的 Gas 优化技巧：

1. 使用局部变量：在合约中使用局部变量比使用状态变量更加高效，因为局部变量不需要存储在以太坊上。
2. 避免循环：尽可能避免使用循环，因为循环会增加 gas 消耗。如果必须使用循环，可以考虑使用 for 循环而不是 while 循环。这是因为for循环有一个特殊的变量i，它的生命周期仅限于循环体内。当循环结束后，i变量也会被销毁。而while循环中使用的变量需要在循环结束后手动销毁，这将消耗更多的Gas。
3. 使用 uint256 类型：尽可能使用 uint256 类型，因为它比其他类型更加高效。
4. 避免使用复杂的数据结构：尽可能避免使用复杂的数据结构，如映射和结构体，因为它们会增加 gas 消耗。
5. 使用库函数：尽可能使用库函数，因为库函数已经过优化，可以减少 gas 消耗。
6. 避免重复调用外部合约：尽可能避免在循环中重复调用外部合约，因为每次调用都需要消耗一定的 gas。

   调用外部合约的函数会有额外的Gas消耗。这是因为在调用外部合约函数时需要发起一个新的交易或者创建一个新的消息，这会涉及到新的存储和计算成本。具体而言，调用外部合约的函数需要支付的Gas包括：

   1. 调用消耗的Gas，即执行代码的基本费用；
   2. 存储数据的Gas，包括输入参数和返回值；
   3. 计算成本的Gas，即代码执行所需的Gas。

      可以将执行代码的基本费用看作是在每次函数调用时必须支付的固定费用，而计算成本的 gas 则是根据代码的实际执行情况动态计算的。因此，执行简单代码的函数调用将消耗较少的 gas，而执行复杂代码的函数调用将消耗更多的 gas。

   需要注意的是，如果调用的外部函数是`view`或`pure`函数，那么只会进行本地调用而不会发生交易或者创建新消息，因此不会产生额外的Gas消耗。

1. 避免重复计算：尽可能避免重复计算同一项值，这样可以减少不必要的计算和存储消耗。
2. 合理使用注释：添加合适的注释可以帮助他人更好地理解代码，并且可以减少不必要的代码。
3. 使用批量处理：使用批量处理可以减少交易数量，从而减少 gas 消耗。

   e.g. 打包多个转账操作到一个交易中可以使用如下方式：

```
function batchTransfer(address[] memory recipients, uint256[] memory amounts)
public {
    require(recipients.length == amounts.length, "Invalid input");
    uint256 totalAmount = 0;
    for (uint i = 0; i < amounts.length; i++) {
        totalAmount += amounts[i];
    }
    require(totalAmount <= address(this).balance, "Insufficient balance");
    for (uint i = 0; i < recipients.length; i++) {
        recipients[i].transfer(amounts[i]);
    }
}
```

上述代码中，`batchTransfer` 函数接受两个参数，一个是接收者地址数组，另一个是对应的转账金额数组。该函数首先检查传入的参数是否合法，然后计算所有转账金额的总和，以确保合约余额足够进行转账。最后，使用一个循环逐一向每个接收者地址发送对应的转账金额。

总之，优化 gas 消耗是 Solidity 开发中非常重要的一环。在编写智能合约时，需要遵循以上 Gas 优化技巧，以最大程度地减少 gas 消耗，从而降低成本。

- https://docs.soliditylang.org/en/develop/types.html
- https://docs.soliditylang.org/en/develop/units-and-global-variables.html
- https://docs.soliditylang.org/en/develop/contracts.html
- https://docs.soliditylang.org/en/develop/cheatsheet.html