一些功能索引

.push 方法是用于在Solidity中向动态数组添加元素的便捷方法，让数组长度可以动态增长。

它允许在数组的末尾添加一个新元素。这在需要动态增长数组长度的情况下非常有用，因为在Solidity中，静态数组（Static Array）的长度是固定的，而动态数组的长度可以随着元素的添加而增长。

// 声明一个动态数组

uint[] public myArray;

function addElement(uint value) public {

// 使用.push方法在数组末尾添加一个元素

myArray.push(value);

}

2-4 solidity基础—函数

·每个合约都有一个地址，就像每个账户都有地址一样。

·我们在区块链上做任何事情修改任何状态，就是发送一次交易，部署一个合约就修改了区块链，让链上拥有这个合约。

·部署合约需要一些gas

·函数变量的4种可见度标识符

Pubic：visible externally and internally(create a getter function for storage/state variables)

内外都看得见，任何与合约交互的人，都可以看到它的值

Private: only visible in the current contract

合约唯一可以调用这个变量

External:只对合约外部可见，表示合约外的账户可以调用这个函数

Internal:只有这个合约或者继承它的合约可以读取

如果在变量定义的时候没有标注，会默认为internal

·智能合约里的函数功能越多，这个函数消耗的计算量就更多

·solidity中有两个关键字，标志函数的调用不需要消耗gas:

view:只会读取这个合约的状态，不允许修改任何状态，

Pure:不允许读取区块链数据，也不允许修改任何状态

执行它们的时候，不发送交易，我们只是在链下读取数值。

但是用retrieve函数调用就要消耗gas

2-5 solidity基础—数组和结构体

·Array：数组是存储列表，或者说是储存一系列对象的一种方式。

先声明对象的类型，然后是对象的可见性，最后是变量名

People[] public people;

Dynamic array :动态数组，初始化的时候没有规定它的大小

   struct People{

        uint256 favoriteNumber;

        string name;

    }

定义结构体：People

People[] public people;

创建由People结构组成的数组people变量

    function addPerson(string memory \_name, uint256 \_favoriteNumbe)public {

        People memory newPerson = People({favoriteNumber:\_favoriteNumbe , name:\_name});

        people.push(newPerson);

    }

动态添加数据进数组

定义newperson 接受传入的参数，再更新到people.push

有很多方法，比如直接在people.push里传参

    function addPerson(string memory \_name, uint256 \_favoriteNumbe)public {

        people.push(People(\_favoriteNumbe,\_name));

    }

2-6 solidity基础—错误和警告

红色—错误

橙色—警告

2-6 solidity基础—Memory，Storage，Calldate 的介绍

·函数参数的数据位置（data location)

目前solidity有六种方式可以存储数据，重点学习其中三个

Calldata,memory:这个变量只是暂时的存在

calldate是不能被修改的临时变量

Memory是可以被修改的临时变量

Storage:甚至存在于正在执行的函数外，是可以被修改的永久变量

这六个数据位置只适用于数组，结构或映射类型，不能是unit

因为数组，结果和映射在solidity中被认为是特殊的类型，而solidity可以自动知道unit256的位置。

所以，结构体(string)，映射(function)，数组(array)在作为参数被添加到不同函数时，需要给变量加一个数据位置。

2-8 solidity基础—Mappings

Mapping—映射想象成字典，它是一组键值对，每对key键返回与该键关联的某个value值。

Mapping(string=>unit256) public nameToFavoriteNumber;

然后记得将在动态添加变量的地方将favoritenumber传入nameToFavoriteNumber[\_name]

2-9 部署第一部合约

2-10 总结

证书说明

1 确定使用版本program solidity ^0.8.8;

2 创建合约对象并命名 contract相当于其他编程语言里的class

3 solidity很多的数据类型

4 view和pure不修改区块链状态的函数

5 数据位置(data location)

Calldata:不能被修改的临时变量

Memory:能被修改的临时变量

Storage: 永久存在

3-2引入其他合约

·先完整复制一个合约内容，然后

contract StorageFactory{

SimpleStorage public simpleStorage;

Function createSimpleStorageContract() public{

simpleStorage = new SimpleStorage;

}

}

为了简化复制这一个繁琐的过程，我们import”./SimpleStorage.sol”直接引入这个合约

3-3 与其他合约交互

·要与其他合约进行交互，我们需要两样东西：1 合约地址 2 合约ABI

ABI是（application binary interface）应用程序二进制接口，告诉我们合约代码如何进行交互

在Complication Details 里可以查找到ABI

3-4 继承与重载

让ExtraStorage继承SimpleStorage的全部函数

Import “./SimpleStorage.sol”

Contract ExtraStorage is SimpleStorage{

}

现在就能确保这个ExtraStorage是SimpleStorage的子函数，它继承了SimpleStorage的所有功能。

重载

当子合约要直接用父合约的函数时，要在父合约对应的函数加一个virtual

在父合约引用函数的时候加一个 override

第三章总结

·用import导入其他合约

·用A a =new A()建立新的合约

·只要获得合约的地址和ABI就能和其他合约进行交互

·建立子合约继承（is）父合约

·子合约（virtual）重载父合约（override）的函数

4-1 FundMe合约

·每次部署一个合约时，就可以获得一个合约地址，它与钱包地址几乎一致，所有钱包和合约都可以持有像ETH这样的原生区块链通证。

·区块链是确定性系统，这意味着它们自身不能与现实世界的数据和事件交互。

Chainlink:它是一种工具，给只能合约提供外部的数据或计算

如果使用中心化的预言机，将重新引入单点失败风险，会违背构建只能合约的初衷，

所以我们不想通过中心化节点获取我们的数据或进行外部计算。

如果调用HTTP的API，在不同的时间调用节点或者他们做了其他事情，都会发送无法共识。

Chainlink是一个模块化，去中心的网络，可以定制以传输任何数据或者执行你想要的外部计算。

·date feed

·chainlink VRF(可验证的随机数函数)

·chainlink keepers(通过event触发的函数执行)

·connet to api

接口和喂价

调用外部合约

1 引入合约（可以从很多地方引入）

2 获得合约的address

3      function getVersion() public view returns (uint256){

        AggregatorV3Interface priceFeed =AggregatorV3Interface(0x694AA1769357215DE4FAC081bf1f309aDC325306);

        return priceFeed.version();

    }

4-3 solidity基础—数组和结构体

Msg.sender(address):交易或组合信息的发送者

Msg.value(uint):交易中发送的wei数量

Libraries(库)

现在所有在库中的函数都必须是internal,我们将会使用在库中的不同函数，可以被uint 256调用。

Msg.sender.A;

A(msg.sender);

这两种表达方式是一样的

4-4 SafeMath库，overflow checking，”unchecked”关键词

uint8 public bignumber = 255;

    function add() public {

        bignumber = bignumber + 1;

    }

在0.8之前的版本，无符号整型和整型是运行在unchecked这个概念下的，意味着如果你超过一个数字的上限，它只会绕回去并从最低的数字开始

在0.8版本之后，系统会自动检查，如果超过，它会交易失败，我们仍然得到的255值

如果想使用循环 则unchecked {bignumber = bignumber +1;}

不过，unchecked可以节约gas，如果确定数字不会超过上限，就可以用unchecked节约gas.

4-5 For Loop

我们希望项目方能从合约中提取资金

For循环：是一种将某种类型的索引对象进行循环的方法或者只是将一项任务重复执行某个次数

For(starting index ; ending index ; step index){

code

}

for (uint256 funderIndex = 0; funderIndex < funders.length; funderInde++){

            address funder = funders[funderIndex];

            //从 funders 数组中获取索引为 funderIndex 处的地址。

            //将获取的地址赋值给 funder 变量，从而将该地址存储在 funder 中。

            addressToAmountFunded[funder] = 0;

            //提取这个funder的资金后将它的value清零

        }

4-6 Resetting an Array重置数组

//重置数组

        funders = new address[](0);

        //全新的address数组并且里面没有任何（0个）对象

从里面正真的提取资金

三种方式：transfer , send , call

msg.senter.transfer(address(this).balance);

将this这个整个智能合约中剩余的钱都transfer到msg.value这个目的地

msg.value 是一个全局变量，用于获取当前交易的以太币（Ether）价值，以wei为单位。它只能在函数调用期间使用，并且只在函数是由其他合约或外部账户发送以太币的交易中才会有值。

注意，这里的msg.senter是地址address变量，不具有支付payable的功能，

payable 类型则是一个特殊的类型标记，表示一个智能合约可以接收以太币。在默认情况下，智能合约是不接收以太币的，即没有 payable 关键字的合约只能执行纯计算操作

进一步：

 //transfer:

        payable (msg.sender).transfer(address(this).balance);

但是

Transfer(2300gas,throws error)

上限是2300gas，如果超过了就会滚回结束交易

Send(2300gas,return bool)

上限也是2300gas，如果超过了就会返回一个表示运行是否成功的bool值

require(condition, errorMessage);

condition: 这是一个布尔表达式或返回布尔值的函数，用于表示某个条件是否满足。如果条件为 false，则触发 require 并回滚交易。

       // bool sendSuccess = payable (msg.sender).send(address(this).balance);

        //require(sendSuccess,"Send failed");

Call(forward all gas or set gas , returns bool);

Call在solidity中更常用

//call(推荐)

        (bool callSuccess, ) = payable (msg.sender).call{value:address(this).balance}("");

        require(callSuccess , "Call failed");

4-7solidity—构造函数

我们现在可以让任何人捐款，但是也让任何人提款，我们只想让募集者真正的提取资金

当你部署这个合约时，构造函数会被自动调用，并根据你传递的参数值来初始化合约

名称和合约名相同：构造函数的名称必须与合约的名称完全相同，这样在合约部署时才能自动调用构造函数。

没有返回类型：构造函数没有返回类型，包括不带 returns 关键字。

不可调用：构造函数在合约部署时自动调用一次，但在部署后不能手动调用。

只执行一次：每个合约只能有一个构造函数，并且它仅在合约部署时执行一次，用于初始化合约的状态和变量。

    //构造函数

    address public owner;

    constructor (){

        owner = msg.sender;

        //让owner等于msg.sender,这个msg.sender就是部署这个合约的人

    }

4-8 solidity—Modifiers

双等号是用来检查这两个变量是否等价

require(msg.sender == owner,"Sender is not owner!");

这样保证了这个withdraw合约只能被合约的拥有者所调用

但是当一个合约有很多函数时，我们不想反复复制这一行到所有函数里

这时就需要我们的（modifier）

 function withdraw() public onlyowner{。。。。}

modifier onlyowner{

        require(msg.sender == owner,"Sender is not owner!");

        \_;

    }

运行到withdraw时发现有onlyowner，优先运行onlyowner：先满足require再运行withdraw中剩余代码。

4-11 solidity进阶—Immutable(常量) & Constant(不可变的)

 uint256 public constant MINIMUM\_USD = 50 \* 1e18;//设置最小金额

当加上constant词后，这个minimumusd将不再展映一个存储空间，更容易被读取

一般常量变量用大写表述

address public immutable i\_owner;

Owner的地址时不可变的，所以标记为immutable

一般不可变量前面加上i\_

4-12 solidity进阶—Custom Error(自定义错误类型)

之前使用require语句定义错误语句会增加很多strings

现在可以自定义错误

先在开头

error NotOwner();

再在需要判断报错的地方revert

 modifier onlyowner{

        //require(msg.sender == i\_owner,"Sender is not owner!");

        if (msg.sender != i\_owner){

            revert NotOwner();

        }

        \_;

    }

revert() 是一个异常处理函数，用于中止当前的合约执行并回滚所有的状态更改。

revert() 函数不接受任何参数，但可以用来传递错误消息。当触发 revert() 时，当前的交易将被中止，并返回所有以太币和燃气费用给交易的发送者。这可以确保不会因为错误而浪费用户的以太币。

4-13 solidity进阶—Receive & Fallback

如果我们不调用fund函数直接发送ETH呢？

我们将优化代码帮助那些不小心调用了错误函数（没用fund函数）或者不小心给这个合约转钱，这样它们虽然多交一些gas费但是地址和记录都会存档

receive() 函数只能接收以太币，如果尝试在调用过程中发送以太币，会触发异常并导致交易失败。

其实我们使用其他函数来接受传送来的ETH也是用某种数据填充这个calldata值，与receive的功能一样。

每当数据和交易一起被发送的时，solidity：既然你在发送数据，那你不是在寻找receive，你在寻找某个函数，但是当它发现没有任何符合发送数据的函数时，就去寻找fallback函数。

而fallback函数，当数据和交易同时发送的时候也会被触发。

// Explainer from: https://solidity-by-example.org/fallback/

// Ether is sent to contract

// is msg.data empty?

// / \

// yes no

// / \

// receive()? fallback()

// / \

// yes no

// / \

//receive() fallback()

5-2 软件安全和环境准备

只用remix有很多弊端，不能真正地与项目其他部分集成，他对测试和部署的支持有限，下面我们要开始使用Hardhat这就是所谓的智能合约开发框架，而它是基于javascript的开发环境

gitpod是一种云开发环境，它可以让你在远程服务器上运行你的代码，只适合在浏览器上或者是以连接到其他服务器的方式去运行visual studio code

tanggaga@DESKTOP-TBCNT04:~/folder$ nvm --version

0.39.1

tanggaga@DESKTOP-TBCNT04:~/folder$ nvm install 16.14.2

v16.14.2 is already installed.

Now using node v16.14.2 (npm v8.5.0)

tanggaga@DESKTOP-TBCNT04:~/folder$ node --version

v16.14.2

这里显示nvm 和node安装完毕！

mkdir是在Bash终端中用于创建新文件夹（目录）的命令。

创建一个名为my\_folder的新目录：

mkdir my\_folder

用cd进入文件里

Cd my\_folder

tanggaga@DESKTOP-TBCNT04:~$ mkdir 8\_7

tanggaga@DESKTOP-TBCNT04:~$ cd 8\_7

tanggaga@DESKTOP-TBCNT04:~/8\_7$ mkdir ethers-simple-storage

tanggaga@DESKTOP-TBCNT04:~/8\_7$ ls

ethers-simple-storage

tanggaga@DESKTOP-TBCNT04:~/8\_7$ code ethers-simple-storage/

tanggaga@DESKTOP-TBCNT04:~/8\_7$

5-8 Javascript 异步编程介绍

我们在使用hardhat写脚本的时候我们不使用常规函数，而是使用异步函数(async function)

同步

1 玉米放进微波炉 ——promise

2 等待爆米花制作完成

3 给爆米花撒盐

异步

1 玉米放进微波炉

2 给客人倒水

3

Promise

可以是”pending”(挂起的)，”fulfilled”(已成功的)，”rejected”(已失败的)

爆米花在制作的时候会返回一个promise，通过这个告诉代码爆米花的完成情况

async function dianying(){

    await cookpopcorn()

    await pourdrinks()

    startmovie()

}

function cookpopcorn(){

    return Promise()

}

当我们要部署合约时，我们必须要等合约部署完成

如果不用异步，在它还没部署的时候就尝试运行合约上的代码，是行不通的

5-9 编译Solidity

Package.json是一个描述我们的项目以及它的依赖关系文件

Yarn是一个流行的包管理器，用于管理JavaScript代码的依赖项（也就是JavaScript包）

使用Yarn进行包管理通常涉及以下几个步骤：

1 在项目根目录下创建一个package.json文件，它包含了项目的依赖信息和一些配置。

2 使用Yarn的命令行工具在项目中安装依赖包：yarn add package-name。

3 Yarn会将依赖包下载并安装到项目的node\_modules目录下，并自动更新package.json文件。

4 通过yarn命令来运行项目或其他脚本。

"scripts": {

    "compile": "yarn solcjs --bin --abi --include-path node\_modules/ --base-path . -o . SimpleStorage.sol"

  }

在这里加上scripts命令相当于在remix里的compile快捷键

5-10 Ganache以及网络介绍

我们需要一个假的区块链，我们需要Ganache的工具，它和remix里的虚拟机很相似

RPC URL:到某个人正在运行的区块链节点的连接

这个URL连接我们对API的调用以及而我们与区块链节点的交互

RPC:远程过程调用

URL:统一资源定位符

当我们有这个端点，就可以开始对这个端点进行API调用了

5-11 Ethers.js介绍

Ethers是支撑Hardhat环境的而主要工具

安装：yarn add ethers

将ethers导入脚本deploy.js : const ethers = require("ethers");

const provider = new ethers.JsonRpcApiProvider("http://127.0.0.1:7545");

连接到这个URL去，这部分代码使用 new 关键字创建了一个新的 ethers.JsonRpcApiProvider 的实例。通过传递 JSON-RPC URL 参数，它初始化了一个连接到指定节点的提供者对象。

获取钱包的具体方式是：

const wallet = new ethers.Wallet();

Wallet函数需要接受几个参数，比如私钥和provider

注意，直接把私钥粘贴进代码是一个大忌！不过现在没关系

const provider = new ethers.JsonRpcApiProvider("http://127.0.0.1:7545");

  const wallet = new ethers.Wallet(

    "0xacd0532685b1bf29f8a5462cd0cefc4c81d620be11f2fe0dac3b38be29308b4d",

    provider

  );

这两行提供我们与智能合约交互所需要的一切

一个提供了我们与区块链的连接

一个提供了带有私钥的钱包

以使我们可以对不同的交易进行签名

为了部署我们的合约，我们需要智能合约的ABI以及编译后的BIN（二进制代码）

我们需要读取这两个文件，需要一个叫 fs 的包

const fs = require("fs-extra");

const abi = fs.readFileSync("./SimpleStorage\_sol\_SimpleStorage.abi","utf8");

  //sync:同步读取

  //uft8是我们用来对这个文件进行编码的方式

  const binary = fs.readFileSync("./SimpleStorage\_sol\_SimpleStorage.bin","utf8");

合约工厂是一个用来部署合约的对象

具体写法是：

const contractFactory = new ethers.ContractFactory(abi,binary,wallet);

Abi:代码能知道如何让与合约进行交互

Binary：编译后的主代码

Wallet：提供了一个私钥能使我们能够用它的签名并部署这个合约

console.log 是一个用于在控制台（Console）输出信息的 JavaScript 函数。它通常用于调试和开发过程中，可以将信息输出到浏览器的开发者工具控制台，或者在 Node.js 环境中输出到终端。

然后我们就可以用enthers部署这个合约了

 const contract = await contractFactory.deploy();

用await告诉代码等一下，先等合约部署成功再运行

Await关键字意味着这里将处理promise

而 contractFactory就会在这里等待返回一个contract对象

const ethers = require("ethers");

const fs = require("fs-extra");

async function main() {

  //http://172.23.80.1:7545

  const provider = new ethers.JsonRpcProvider("http://172.23.80.1:7545");

  const Wallet = new ethers.Wallet(

    "0x622e95a8f7ee7c255141275b85a953525f84a5d171c11d46af0f4f73947672b1",

    provider

  );

  const abi = fs.readFileSync("./SimpleStorage\_sol\_SimpleStorage.abi", "utf-8");

  const binary = fs.readFileSync(

    "./SimpleStorage\_sol\_SimpleStorage.bin",

    "utf-8"

  );

  const contractFactory = new ethers.ContractFactory(abi, binary, Wallet);

  console.log("正在部署中，等一下");

  const contract = await contractFactory.deploy();

  console.log(contract);

}

main()

  .then(() => process.exit(0))

  .catch((error) => {

    console.error(error);

    process.exit(1);

  });

5-12 交易重写项

"deploy" 是指部署智能合约（Smart Contract）到区块链网络上的一个过程。

点击这个 按ctril就可以进入一个函数并查看它在哪里被定义

通过const contract = await contractFactory.deploy();

  const deploymentReceipt = await contract.deploymentTransaction().wait(1);

  //.wait(1) 是一个异步函数，它等待指定的事务完成。

  //在这里，1 是指等待第一个确认的区块。通常，交易在多个确认之后才被视为最终确认。这可以确保交易已被网络广泛接受并确认。

  console.log(deploymentReceipt);

我们可以在终端看到所有交易过程

交易回执（Transaction Receipt）是一个重要的概念。它是在执行某个交易（例如合约部署、转账等）后，区块链网络返回给你的信息，用于确认该交易的状态和详细信息。

只有在等待区块链确认时才能获得交易回执(transaction receipt)

否则只有合约对象以及包含的部署交易的信息

**交易回执时你等待区块确认的时候获得的（回执是完成后的回执，说明部署成功）**

**而部署交易或者说是交易响应是创建交易的时候获得的（响应是创建的时候就有了，有可能还在pending)**

**9-1**

const ethers = require("ethers");

const fs = require("fs-extra");

async function main() {

  const provider = new ethers.JsonRpcProvider("http://127.0.0.1:8545");

  const wallet = new ethers.Wallet(

    "0x6ef5b9e94bd6f9ae1548fb2007b9205a3391cf9305902902927e165b4465fd80",

    provider

  );

  //连接测试网

  const abi = fs.readFileSync("./SimpleStorage\_sol\_SimpleStorage.abi", "utf8");

  const binary = fs.readFileSync("SimpleStorage\_sol\_SimpleStorage.bin", "utf8");

  //连接sol文件

  // const contractFactory = new ethers.ContractFactory(abi, binary, wallet);

  // console.log("正在部署中。。。");

  // const contract = await contractFactory.deploy();

  // const transactionReceipt = await contract

  //   .deploymentTransaction({ gasLimit: 3000000 })

  //   .wait(1);

  //.wait(1) 是一个异步函数，它等待指定的事务完成。

  //在这里，1 是指等待第一个确认的区块。通常，交易在多个确认之后才被视为最终确认。这可以确保交易已被网络广泛接受并确认。

  console.log("下面只传输交易信息，只是实际部署或仅用交易数据发送交易的方式");

  const tx = {

    nonce: 2,

    gasPrice: 2000000000,

    gasLimit: 30000000,

    to: null,

    value: 0,

    data: "",

    chainId: 1337,

  };

  const signedTxResponse = await wallet.signTransaction(tx);

  console.log(signedTxResponse);

  console.log("下面是我们的交易过程");

  console.log(contract.deploymentTransaction);

  console.log("下面是我们的交易回执");

  console.log(transactionReceipt);

  //异步，保证部署完成之后再下一步行动

}

main()

  .then(() => process.exit(0))

  .catch((error) => {

    console.error(error);

    process.exit(1);

  });

//异步过程

**5-14 通过ether.js发送\_raw\_交易**

0xe26511F6dD23346c2830b87B400d402b8CF292c1

7f7c89d3c5cc42b5b6553a0a127c476d

"0xa469F39796bE2E51117693880dB3E6438d"

yarn solcjs lambda.sol --bin --abi -o ~/8\_7/ethers-simple-storage/output\_lambda

yarn solcjs chainlink/chainlink/docs.chain.link/samples/APIRequests/ATestnetConsumer.sol --bin --abi -o chainlink/chainlink/docs.chain.link/samples/APIRequests/output

Orcal:0x7403bD34860eDda74d01f81085905Bb28aD85f78

Job: 6a80365f9c6743da-bef94515f815441c