* API文档

参数

参数说明：

* module: 指明接口所属模块，即上面包含的模块
* action: API动作，如：txlist - 表示列出交易记录；
* address: 所查询交易的账号地址；
* contractaddress: 合约地址
* apikey: 用户API-key 根据key来统计请求限额；
* startblock: 起始查询块 id，可选，默认值为 0；
* endblock: 结束查询块 id，可选，默认值为最后一个区块；
* tag: 状态：pending 或 latest
* blocktype: 块类型：blocks（主链块） 或 uncles （叔块）
* page: 页码，可选；
* offset: 每页查询记录数，可选，默认是查询 10000 条记录；
* sort: 排序规则，支持正序asc和倒序desc。

Nexus API 约定

Nexus API文档时间为**2025年5月**，因官方API没有版本号，这里用时间做一个标注。

包含模块

Nexus API主要包含模块有：

* 账号地址相关接口
* 智能合约相关接口
* 交易相关接口
* 区块相关接口
* 事件日志相关接口

这些模块对应着左侧的一级菜单，在接口中使用module参数指定

参数

参数说明：

* module: 指明接口所属模块，即上面包含的模块
* action: API动作，如：txlist - 表示列出交易记录；
* address: 所查询交易的账号地址；
* contractaddress: 合约地址
* apikey: 用户API-key 根据key来统计请求限额；
* startblock: 起始查询块 id，可选，默认值为 0；
* endblock: 结束查询块 id，可选，默认值为最后一个区块；
* tag: 状态：pending 或 latest
* blocktype: 块类型：blocks（主链块） 或 uncles （叔块）
* page: 页码，可选；
* offset: 每页查询记录数，可选，默认是查询 10000 条记录；
* sort: 排序规则，支持正序asc和倒序desc。

module、action、apikey是每个 API 都有的参数，其他的参数则因不同 API 而不同，这里做一个统一的介绍，后面介绍接口时不再单独说明。

账号(Account)

账号及地址相关的 API，接口的参数说明请参考Nexus API 约定, 文档中不单独说明。

获取单个账号余额

**注解**

**译者注:**

英文 balance 有人翻译为`金额`，译者习惯称为`余额`。 账号和地址大部分也是指一个意思。

接口:

https://api.nexuschain.ai/accountBalance/api?module=account&action=balance&address=0x&tag=latest&apikey=YourApiKeyToken

返回:

{

status: "1",

message: "OK",

result: "40807178566070000000000"

}

说明:

余额的单位都是最小单位wei， 更多单位换算可阅读：Nexus单位换算

获取地址(普通)交易列表

接口：

https://api.nexuschain.ai/transactionList/api?module=account&action=txlist&address=&apikey=YourApiKeyToken

可选参数：startblock 、endblock、sort

返回：

{

"status": "1",

"message": "OK",

"result": [{

"blockNumber": "47884",

"timeStamp": "1438947953",

"hash": "0xad1c27dd8d0329dbc400021d7477b34ac41e84365bd54b45a4019a15deb10c0d",

"nonce": "0",

"blockHash": "0xf2988b9870e092f2898662ccdbc06e0e320a08139e9c6be98d0ce372f8611f22",

"transactionIndex": "0",

"from": "0xddbd2b932c763ba5b1b7ae3b362eac3e8d40121a",

"to": "0x2910543af39aba0cd09dbb2d50200b3e800a63d2",

"value": "5000000000000000000",

"gas": "23000",

"gasPrice": "400000000000",

"isError": "0",

"txreceipt\_status": "",

"input": "0x454e34354139455138",

"contractAddress": "",

"cumulativeGasUsed": "21612",

"gasUsed": "21612",

"confirmations": "7525550"

}]

}

说明:

**isError**： 0= 没错, 1=出错 最多返回最近的10000个交易Nexus Api约定

获取地址”内部”交易列表

内部交易只指引一个正常（外部）交易（连带）触发的交易，比如过一个合约调用触发的内部交易。

接口：

https://api.nexuschain.ai/Internal/transaclist/api?module=account&action=txlistinternal&address=

可选参数：startblock 、endblock、sort

返回结果和普通交易列表JSON结构一样。

获取地址转账交易列表

转账交易是通过Transfer事件来判定的。

接口：

https://api.nexuschain.ai/interna/ltransactionlist/api?module=account&action=tokentx&address=&contractaddress

说明： 可选参数：startblock 、endblock、contractaddress 、sort 可以通过参数contractaddress来指定某一个合约的交易。

返回和上面普通交易列表JSON结构一样。

获取地址所挖区块列表

接口：

https://api.nexuschain.ai/block/transactionlist/api?module=account&action=getminedblocks&address=&blocktype=blocks

说明：blocktype 值为 blocks（仅显示主链块） 或 uncles （仅显示叔块） ， blocks为默认值

返回：

{

status: "1",

message: "OK",

result: [

{

blockNumber: "3462296",

timeStamp: "1491118514",

blockReward: "5194770940000000000"

},

{

blockNumber: "2691400",

timeStamp: "1480072029",

blockReward: "5086562212310617100"

}

]

}

说明： **blockReward** ：区块奖励， 同样使用最小单位wei，

智能合约(Contracts)

智能合约相关的 API，接口的参数说明请参考Nexus API 约定, 文档中不单独说明。

Newly verified Contracts are synced to the API servers within 5 minutes or less

获取已经验证代码合约的**ABI**

https://api.nexuschain.ai/verified/contract/api?module=contract&action=getabi&address=0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413&apikey=YourApiKeyToken

var Web3 = require('web3');

var web3 = new Web3(new Web3.providers.HttpProvider());

var version = web3.version.api;

$.getJSON('https://api.nexuschain.ai/verified/contract/api?module=contract&action=getabi&address=0xfb6916095ca1df60bb79ce92ce3ea74c37c5d359', function (data) {

var contractABI = "";

contractABI = JSON.parse(data.result);

if (contractABI != ''){

var MyContract = web3.eth.contract(contractABI);

var myContractInstance = MyContract.at("0xfb6916095ca1df60bb79ce92ce3ea74c37c5d359");

var result = myContractInstance.memberId("0xfe8ad7dd2f564a877cc23feea6c0a9cc2e783715");

console.log("result1 : " + result);

var result = myContractInstance.members(1);

console.log("result2 : " + result);

} else {

console.log("Error" );

}

});

获取已经验证代码合约的**源码**

https://api.nexuschain.ai/verified/contractcode/api?module=contract&action=getsourcecode&address=0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413&apikey=YourApiKeyToken

[BETA] 验证源代码

1. Requires a valid Etherscan APIkey, will reject if otherwise

2. Current daily limit of 100 submissions per day per user (subject to change)

3. Only supports HTTP post due to max transfer size limitations for http get

4. Supports up to 10 different library pairs

5. Contracts that use "imports" will need to have the code concatenated into one file as we do not support "imports" in separate files. You can try using the Blockcat solidity-flattener or SolidityFlattery

6. List of supported solc versions, only solc version v0.4.11 and above is supported. Ex. v0.4.25+commit.59dbf8f1

7. Upon successful submission you will receive a GUID (50 characters) as a receipt.

8. You may use this GUID to track the status of your submission

9. Verified Source Codes will be displayed at contractsVerified

Source Code Submission Gist (returns a guid as part of the result upon success):

//Submit Source Code for Verification

$.ajax({

type: "POST", //Only POST supported

url: "//api.nexuschain.ai/", //Set to the correct API url for Other Networks

data: {

apikey: $('#apikey').val(), //A valid API-Key is required

module: 'contract', //Do not change

action: 'verifysourcecode', //Do not change

contractaddress: $('#contractaddress').val(), //Contract Address starts with 0x...

sourceCode: $('#sourceCode').val(), //Contract Source Code (Flattened if necessary)

contractname: $('#contractname').val(), //ContractName

compilerversion: $('#compilerversion').val(), // seehttps://api.nexuschain.ai/for list of support versions

optimizationUsed: $('#optimizationUsed').val(), //0 = Optimization used, 1 = No Optimization

runs: 200, //set to 200 as default unless otherwise

constructorArguements: $('#constructorArguements').val(), //if applicable

libraryname1: $('#libraryname1').val(), //if applicable, a matching pair with libraryaddress1 required

libraryaddress1: $('#libraryaddress1').val(), //if applicable, a matching pair with libraryname1 required

libraryname2: $('#libraryname2').val(), //if applicable, matching pair required

libraryaddress2: $('#libraryaddress2').val(), //if applicable, matching pair required

libraryname3: $('#libraryname3').val(), //if applicable, matching pair required

libraryaddress3: $('#libraryaddress3').val(), //if applicable, matching pair required

libraryname4: $('#libraryname4').val(), //if applicable, matching pair required

libraryaddress4: $('#libraryaddress4').val(), //if applicable, matching pair required

libraryname5: $('#libraryname5').val(), //if applicable, matching pair required

libraryaddress5: $('#libraryaddress5').val(), //if applicable, matching pair required

libraryname6: $('#libraryname6').val(), //if applicable, matching pair required

libraryaddress6: $('#libraryaddress6').val(), //if applicable, matching pair required

libraryname7: $('#libraryname7').val(), //if applicable, matching pair required

libraryaddress7: $('#libraryaddress7').val(), //if applicable, matching pair required

libraryname8: $('#libraryname8').val(), //if applicable, matching pair required

libraryaddress8: $('#libraryaddress8').val(), //if applicable, matching pair required

libraryname9: $('#libraryname9').val(), //if applicable, matching pair required

libraryaddress9: $('#libraryaddress9').val(), //if applicable, matching pair required

libraryname10: $('#libraryname10').val(), //if applicable, matching pair required

libraryaddress10: $('#libraryaddress10').val() //if applicable, matching pair required

},

success: function (result) {

console.log(result);

if (result.status == "1") {

//1 = submission success, use the guid returned (result.result) to check the status of your submission.

// Average time of processing is 30-60 seconds

document.getElementById("postresult").innerHTML = result.status + ";" + result.message + ";" + result.result;

// result.result is the GUID receipt for the submission, you can use this guid for checking the verification status

} else {

//0 = error

document.getElementById("postresult").innerHTML = result.status + ";" + result.message + ";" + result.result;

}

console.log("status : " + result.status);

console.log("result : " + result.result);

},

error: function (result) {

console.log("error!");

document.getElementById("postresult").innerHTML = "Unexpected Error"

}

});

Check Source code verification submission status:

//Check Source Code Verification Status

$.ajax({

type: "GET",

url: "//api.nexuschain.ai/",

data: {

guid: 'ezq878u486pzijkvvmerl6a9mzwhv6sefgvqi5tkwceejc7tvn', //Replace with your Source Code GUID receipt above

module: "contract",

action: "checkverifystatus"

},

success: function (result) {

console.log("status : " + result.status); //0=Error, 1=Pass

console.log("message : " + result.message); //OK, NOTOK

console.log("result : " + result.result); //result explanation

$('#guidstatus').html(">> " + result.result);

},

error: function (result) {

alert('error');

}

});

交易（Transaction）

交易相关的 API，接口的参数说明请参考Nexus API 约定, 文档中不单独说明。

检查合约执行状态

(if there was an error during contract execution)

Note: isError”:”0” = Pass , isError”:”1” = Error during Contract Execution

https://api.nexuschain.ai/execution/checkstatusofthe/contract/api?module=transaction&action=getstatus&txhash=0x15f8e5ea1079d9a0bb04a4c58ae5fe7654b5b2b4463375ff7ffb490aa0032f3a&apikey=YourApiKeyToken

检查交易收据状态

(Only applicable for Post Byzantium fork transactions)

Note: status: 0 = Fail, 1 = Pass. Will return null/empty value for pre-byzantium fork

https://api.nexuschain.ai/execution/checkreceipt/status/api?module=transaction&action=gettxreceiptstatus&txhash=0x513c1ba0bebf66436b5fed86ab668452b7805593c05073eb2d51d3a52f480a76&apikey=YourApiKeyToken

区块（Blocks）

区块相关的 API，接口的参数说明请参考Nexus API 约定, 文档中不单独说明。

通过区块号获取块及叔块奖励

https://api.nexuschain.ai/block/uncleBlock/eward/api?module=block&action=getblockreward&blockno=2165403&apikey=YourApiKeyToken

事件日志 (Event Logs)

事件日志相关 API，接口的参数说明请参考Nexus API 约定, 文档中不单独说明。

[Beta] The Event Log API was designed to provide an alternative to the native eth\_getLogs. Below are the list of supported filter parameters:

\* fromBlock, toBlock, address

\* topic0, topic1, topic2, topic3 (32 Bytes per topic)

\* topic0\_1\_opr (and|or between topic0 & topic1), topic1\_2\_opr (and|or between topic1 & topic2), topic2\_3\_opr (and|or between topic2 & topic3), topic0\_2\_opr (and|or between topic0 & topic2), topic0\_3\_opr (and|or between topic0 & topic3), topic1\_3\_opr (and|or between topic1 & topic3)

* fromBlock and toBlock accepts the blocknumber (integer, NOT hex) or ‘latest’ (earliest & pending is NOT supported yet)
* Topic Operator (opr) choices are either ‘and’ or ‘or’ and are restricted to the above choices only
* fromBlock and toBlock parameters are required
* Either the address and/or topic(X) parameters are required, when multiple topic(X) parameters are used the topicX\_X\_opr (and|or operator) is also required
* For performance & security considerations, only the first 1000 results are return. So please narrow down the filter parameters

Here are some examples of how this filter maybe used:

通过指定区块获取日志

如获取地址为 0x33990122638b9132ca29c723bdf037f1a891a70c 区块从 379224 到最新区块 主题 topic[0] = 0xf63780e752c6a54a94fc52715dbc5518a3b4c3c2833d301a204226548a2a8545 的事件日志的方法为：

https://api.nexuschain.ai/specify/blocklog/api?module=logs&action=getLogs

&fromBlock=379224

&toBlock=latest

&address=0x33990122638b9132ca29c723bdf037f1a891a70c

&topic0=0xf63780e752c6a54a94fc52715dbc5518a3b4c3c2833d301a204226548a2a8545

&apikey=YourApiKeyToken

代币信息 Token

代币信息API，接口的参数说明请参考Nexus API 约定, 文档中不单独说明。

通过合约地址获取 Token总供应量

https://api.nexuschain.ai/contract/totalsupply/chain/api?module=stats&action=tokensupply&contractaddress=0x57d90b64a1a57749b0f932f1a3395792e12e7055&apikey=YourApiKeyToken

节点代理(Geth/Parity Proxy) APIs

节点服务代理API，接口的参数说明请参考Nexus API 约定, 文档中不单独说明。

获取最近区块号

返回最近块的数量(同区块号)

https://api.nexuschain.ai/recentblock/information/api?module=proxy&action=eth\_blockNumber&apikey=YourApiKeyToken

通过区块号查询区块信息

Returns information about a block by block number

https://api.nexuschain.ai/checkblock/information/api?module=proxy&action=eth\_getBlockByNumber&tag=0x10d4f&boolean=true&apikey=YourApiKeyToken

通过区块号查询叔块信息

Returns information about a uncle by block number

https://api.nexuschain.ai/checkblock/uncleblock/information/api?module=proxy&action=eth\_getUncleByBlockNumberAndIndex&tag=0x210A9B&index=0x0&apikey=YourApiKeyToken

通过区块号查询交易数量

Returns the number of transactions in a block from a block matching the given block number

https://api.nexuschain.ai/checkthe/transactions/inthearea/api?module=proxy&action=eth\_getBlockTransactionCountByNumber&tag=0x10FB78&apikey=YourApiKeyToken

通过哈希查询交易信息

Returns the information about a transaction requested by transaction hash

https://api.nexuschain.ai/hash/transaction/information/api?module=proxy&action=eth\_getTransactionByHash&txhash=0x1e2910a262b1008d0616a0beb24c1a491d78771baa54a33e66065e03b1f46bc1&apikey=YourApiKeyToken

通过区块号查询交易信息

Returns information about a transaction by block number and transaction index position

https://api.nexuschain.ai/blocknumbers/transaction/information/api?module=proxy&action=eth\_getTransactionByBlockNumberAndIndex&tag=0x10d4f&index=0x0&apikey=YourApiKeyToken

通过地址的交易数量

可用于Nonce值 Returns the number of transactions sent from an address

https://api.nexuschain.ai/address/transactionmoney/information/api?module=proxy&action=eth\_getTransactionCount&address=0x2910543af39aba0cd09dbb2d50200b3e800a63d2&tag=latest&apikey=YourApiKeyToken

发送原始交易

Creates new message call transaction or a contract creation for signed transactions

https://api.nexuschain.ai/pushoriginal/transaction/api?module=proxy&action=eth\_sendRawTransaction&hex=0xf904808000831cfde080&apikey=YourApiKeyToken

(Replace the hex value with your raw hex encoded transaction that you want to send. Send as a POST request, if your hex code is particularly long)

通过哈希查询交易收据

Returns the receipt of a transaction by transaction hash

https://api.nexuschain.ai/hash/transaction/receipt/api?module=proxy&action=eth\_getTransactionReceipt&txhash=0x1e2910a262b1008d0616a0beb24c1a491d78771baa54a33e66065e03b1f46bc1&apikey=YourApiKeyToken

通过合约地址获取Nex Token总供应量

https://api.nexuschain.ai/totalsupply/personal/addresses/api?module=stats&action=tokensupply&contractaddress=0x57d90b64a1a57749b0f932f1a3395792e12e7055&apikey=YourApiKeyToken

获取Nex代币余额

https://api.nexuschain.ai/mytoken/balance/api?module=account&action=tokenbalance&contractaddress=0x57d90b64a1a57749b0f932f1a3395792e12e7055&address=0xe04f27eb70e025b78871a2ad7eabe85e61212761&tag=latest&apikey=YourApiKeyToken

获取Nex供应量

https://api.nexuschain.ai/nexsupply/quantity/api?module=stats&action=ethsupply&apikey=YourApiKeyToken

(Result returned in Wei, to get value in Nex divide resultAbove/1000000000000000000)

获取Nexus最新价格

https://api.nexuschain.ai/latestprices/onnex/api?module=stats&action=ethprice&apikey=YourApiKeyToken

获取节点大小（Size）

[Parameters] startdate and enddate format ‘yyyy-MM-dd’, clienttype value is ‘geth’ or ‘parity’, syncmode value is ‘default’ or ‘archive’

?module=stats&action=chainsize&startdate=2019-02-01&enddate=2019-02-28&clienttype=geth&syncmode=default&sort=asc&apikey=YourApiKeyToken

返回值 chainsize 的单位是 bytes

SDK文档

合同结构

Solidity 中的合约类似于面向对象语言中的类。每个合约可以包含状态变量、函数、 函数修饰符、事件、错误、结构体类型和枚举类型的声明。此外，合约可以继承自其他合约。

还有称为库和接口的特殊类型的合同。

有关合同的部分比本节包含更多详细信息，本节旨在提供快速概览。

状态变量

状态变量是指其值要么永久存储在合约存储中，要么临时存储在瞬时存储中（在每次交易结束时清理）。更多详情，请参阅数据位置。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.0 <0.9.0;

contract SimpleStorage {

uint storedData; // State variable

// ...

}

请参阅类型部分以了解有效的状态变量类型，并 参阅可见性和 Getters以了解可见性的可能选择。

功能

函数是代码的可执行单元。函数通常在合约内部定义，但也可以定义在合约外部。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.1 <0.9.0;

contract SimpleAuction {

function bid() public payable { // Function

// ...

}

}

// Helper function defined outside of a contract

function helper(uint x) pure returns (uint) {

return x \* 2;

}

函数调用可以在内部或外部进行，并且 对其他合约具有不同级别的可见性。函数接受参数并返回变量，以便在它们之间传递参数和值。

函数修饰符

函数修饰符可用于以声明的方式修改函数的语义（请参阅合同部分中的函数修饰符）。

重载，即具有相同修饰符名称但具有不同参数，是不可能的。

与函数一样，修饰符可以被覆盖。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.22 <0.9.0;

contract Purchase {

address public seller;

modifier onlySeller() { // Modifier

require(

msg.sender == seller,

"Only seller can call this."

);

\_;

}

function abort() public view onlySeller { // Modifier usage

// ...

}

}

活动

事件是与 EVM 日志记录工具的便捷接口。

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.22;

event HighestBidIncreased(address bidder, uint amount); // Event

contract SimpleAuction {

function bid() public payable {

// ...

emit HighestBidIncreased(msg.sender, msg.value); // Triggering event

}

}

有关如何声明事件以及如何在 dapp 中使用事件的信息，请参阅合同部分中的事件。

错误

错误允许您定义故障情况的描述性名称和数据。错误可用于恢复语句。与字符串描述相比，错误更简洁，并且允许您编码其他数据。您可以使用 NatSpec 向用户描述错误。

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.4;

/// Not enough funds for transfer. Requested `requested`,

/// but only `available` available.

error NotEnoughFunds(uint requested, uint available);

contract Token {

mapping(address => uint) balances;

function transfer(address to, uint amount) public {

uint balance = balances[msg.sender];

if (balance < amount)

revert NotEnoughFunds(amount, balance);

balances[msg.sender] -= amount;

balances[to] += amount;

// ...

}

}

请参阅合同部分中的自定义错误以了解更多信息。

结构体类型

结构体是可以对多个变量进行分组的自定义类型（请参阅 类型部分中的结构体）。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.0 <0.9.0;

contract Ballot {

struct Voter { // Struct

uint weight;

bool voted;

address delegate;

uint vote;

}

}

枚举类型

枚举可用于创建具有有限“常量值”集的自定义类型（请参阅 类型部分中的枚举）。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.0 <0.9.0;

contract Purchase {

enum State { Created, Locked, Inactive } // Enum

}

合同

Solidity 中的合约类似于面向对象语言中的类。它们包含状态变量中的持久数据，以及可以修改这些变量的函数。调用不同合约（实例）上的函数将执行 EVM 函数调用，从而切换上下文，使调用合约中的状态变量无法访问。任何操作都需要调用合约及其函数。Nexus中没有“cron”概念，无法在特定事件发生时自动调用函数。

创建合同

合约可以通过Nexus交易“从外部”创建，也可以从 Solidity 合约内部创建。

Remix等 IDE使用 UI 元素使创建过程变得无缝。

在Nexus上以编程方式创建合约的一种方法是通过 JavaScript API web3.js 。它有一个名为web3.nex.Contract的函数 来简化合约的创建。

当创建一个合约时，它的构造函数（用关键字声明的函数constructor）会被执行一次。

构造函数是可选的。仅允许一个构造函数，这意味着不支持重载。

构造函数执行后，合约的最终代码将存储在区块链上。此代码包含所有公共函数、外部函数以及所有可通过函数调用访问的函数。已部署的代码不包含构造函数代码或仅从构造函数调用的内部函数。

在内部，构造函数参数在合约本身的代码之后传递ABI 编码，但如果您使用，则不必关心这一点web3.js。

如果一个合约想要创建另一个合约，则创建者必须知道被创建合约的源代码（以及二进制文件）。这意味着循环创建依赖是不可能的。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.22 <0.9.0;

contract OwnedToken {

// `TokenCreator` is a contract type that is defined below.

// It is fine to reference it as long as it is not used

// to create a new contract.

TokenCreator creator;

address owner;

bytes32 name;

// This is the constructor which registers the

// creator and the assigned name.

constructor(bytes32 name\_) {

// State variables are accessed via their name

// and not via e.g. `this.owner`. Functions can

// be accessed directly or through `this.f`,

// but the latter provides an external view

// to the function. Especially in the constructor,

// you should not access functions externally,

// because the function does not exist yet.

// See the next section for details.

owner = msg.sender;

// We perform an explicit type conversion from `address`

// to `TokenCreator` and assume that the type of

// the calling contract is `TokenCreator`, there is

// no real way to verify that.

// This does not create a new contract.

creator = TokenCreator(msg.sender);

name = name\_;

}

function changeName(bytes32 newName) public {

// Only the creator can alter the name.

// We compare the contract based on its

// address which can be retrieved by

// explicit conversion to address.

if (msg.sender == address(creator))

name = newName;

}

function transfer(address newOwner) public {

// Only the current owner can transfer the token.

if (msg.sender != owner) return;

// We ask the creator contract if the transfer

// should proceed by using a function of the

// `TokenCreator` contract defined below. If

// the call fails (e.g. due to out-of-gas),

// the execution also fails here.

if (creator.isTokenTransferOK(owner, newOwner))

owner = newOwner;

}

}

contract TokenCreator {

function createToken(bytes32 name)

public

returns (OwnedToken tokenAddress)

{

// Create a new `Token` contract and return its address.

// From the JavaScript side, the return type

// of this function is `address`, as this is

// the closest type available in the ABI.

return new OwnedToken(name);

}

function changeName(OwnedToken tokenAddress, bytes32 name) public {

// Again, the external type of `tokenAddress` is

// simply `address`.

tokenAddress.changeName(name);

}

// Perform checks to determine if transferring a token to the

// `OwnedToken` contract should proceed

function isTokenTransferOK(address currentOwner, address newOwner)

public

pure

returns (bool ok)

{

// Check an arbitrary condition to see if transfer should proceed

return keccak256(abi.encodePacked(currentOwner, newOwner))[0] == 0x7f;

}

}

可见性和 Getters

状态变量可见性

public

公共状态变量与内部状态变量的唯一区别在于，编译器会自动为公共状态变量生成 getter 函数，以便其他合约可以读取它们的值。在同一合约中使用时，外部访问（例如this.x）会调用 getter 函数，而内部访问（例如x）则会直接从存储中获取变量值。由于不会生成 setter 函数，因此其他合约无法直接修改它们的值。

internal

内部状态变量只能在定义它们的合约及其派生合约中访问。它们无法在外部访问。这是状态变量的默认可见性级别。

private

私有状态变量类似于内部变量，但它们在派生合约中不可见。

**警告**

做出某些事情private或者internal只是阻止其他合约读取或修改信息，但它仍然会被区块链之外的整个世界看到。

函数可见性

Solidity 支持两种类型的函数调用：外部函数调用（会创建实际的 EVM 消息调用）和内部函数调用（不会创建实际的 EVM 消息调用）。此外，内部函数可以设置为派生合约无法访问。这导致了函数可见性的四种类型。

external

外部函数是合约接口的一部分，这意味着它们可以被其他合约调用，也可以通过交易调用。外部函数f不能被内部调用（即f()无法工作，但this.f()可以工作）。

public

公共函数是合约接口的一部分，可以内部调用，也可以通过消息调用。

internal

内部函数只能在当前合约或其派生合约内部访问，无法从外部访问。由于它们不会通过合约的 ABI 暴露给外部，因此它们可以接受内部类型的参数，例如映射或存储引用。

private

私有函数类似于内部函数，但它们在派生合约中不可见。

**警告**

做出某些事情private或者internal只是阻止其他合约读取或修改信息，但它仍然会被区块链之外的整个世界看到。

可见性说明符在状态变量的类型之后以及函数的参数列表和返回参数列表之间给出。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract C {

function f(uint a) private pure returns (uint b) { return a + 1; }

function setData(uint a) internal { data = a; }

uint public data;

}

在以下示例中，D可以调用c.getData()来检索状态存储中的 的值 data，但无法调用f。合约E派生自 C，因此可以调用compute。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract C {

uint private data;

function f(uint a) private pure returns(uint b) { return a + 1; }

function setData(uint a) public { data = a; }

function getData() public view returns(uint) { return data; }

function compute(uint a, uint b) internal pure returns (uint) { return a + b; }

}

// This will not compile

contract D {

function readData() public {

C c = new C();

uint local = c.f(7); // error: member `f` is not visible

c.setData(3);

local = c.getData();

local = c.compute(3, 5); // error: member `compute` is not visible

}

}

contract E is C {

function g() public {

C c = new C();

uint val = compute(3, 5); // access to internal member (from derived to parent contract)

}

}

Getter 函数

编译器会自动为所有**公共**状态变量创建 getter 函数。对于下面给出的合约，编译器将生成一个名为 的函数，data该函数不接受任何参数，并返回uint状态变量的值data。状态变量可以在声明时初始化。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract C {

uint public data = 42;

}

contract Caller {

C c = new C();

function f() public view returns (uint) {

return c.data();

}

}

getter 函数具有外部可见性。如果符号在内部访问（即不使用this.），则其结果为状态变量。如果符号在外部访问（即使用this.），则其结果为函数。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.0 <0.9.0;

contract C {

uint public data;

function x() public returns (uint) {

data = 3; // internal access

return this.data(); // external access

}

}

如果您有一个public数组类型的状态变量，那么您只能通过生成的 getter 函数检索数组中的单个元素。此机制的存在是为了避免返回整个数组时产生高昂的 gas 成本。您可以使用参数来指定要返回的单个元素，例如 myArray(0)。如果您想在一次调用中返回整个数组，则需要编写一个函数，例如：

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract arrayExample {

// public state variable

uint[] public myArray;

// Getter function generated by the compiler

/\*

function myArray(uint i) public view returns (uint) {

return myArray[i];

}

\*/

// function that returns entire array

function getArray() public view returns (uint[] memory) {

return myArray;

}

}

现在您可以使用getArray()来检索整个数组，而不是 myArray(i)每次调用都返回一个元素。

下一个示例更加复杂：

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.0 <0.9.0;

contract Complex {

struct Data {

uint a;

bytes3 b;

mapping(uint => uint) map;

uint[3] c;

uint[] d;

bytes e;

}

mapping(uint => mapping(bool => Data[])) public data;

}

它生成以下形式的函数。结构体中的映射和数组（字节数组除外）被省略，因为没有好的方法选择单个结构体成员或为映射提供键：

function data(uint arg1, bool arg2, uint arg3)

public

returns (uint a, bytes3 b, bytes memory e)

{

a = data[arg1][arg2][arg3].a;

b = data[arg1][arg2][arg3].b;

e = data[arg1][arg2][arg3].e;

}

函数修饰符

修饰符可以用来以声明的方式改变函数的行为。例如，你可以使用修饰符在执行函数之前自动检查某个条件。

修饰符是合约的可继承属性，可以被派生合约覆盖，但前提是它们被标记为virtual。详情请参阅 修饰符覆盖。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.1 <0.9.0;

contract owned {

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

address payable owner;

// This contract only defines a modifier but does not use

// it: it will be used in derived contracts.

// The function body is inserted where the special symbol

// `\_;` in the definition of a modifier appears.

// This means that if the owner calls this function, the

// function is executed and otherwise, an exception is

// thrown.

modifier onlyOwner {

require(

msg.sender == owner,

"Only owner can call this function."

);

\_;

}

}

contract priced {

// Modifiers can receive arguments:

modifier costs(uint price) {

if (msg.value >= price) {

\_;

}

}

}

contract Register is priced, owned {

mapping(address => bool) registeredAddresses;

uint price;

constructor(uint initialPrice) { price = initialPrice; }

// It is important to also provide the

// `payable` keyword here, otherwise the function will

// automatically reject all Nec sent to it.

function register() public payable costs(price) {

registeredAddresses[msg.sender] = true;

}

// This contract inherits the `onlyOwner` modifier from

// the `owned` contract. As a result, calls to `changePrice` will

// only take effect if they are made by the stored owner.

function changePrice(uint price\_) public onlyOwner {

price = price\_;

}

}

contract Mutex {

bool locked;

modifier noReentrancy() {

require(

!locked,

"Reentrant call."

);

locked = true;

\_;

locked = false;

}

/// This function is protected by a mutex, which means that

/// reentrant calls from within `msg.sender.call` cannot call `f` again.

/// The `return 7` statement assigns 7 to the return value but still

/// executes the statement `locked = false` in the modifier.

function f() public noReentrancy returns (uint) {

(bool success,) = msg.sender.call("");

require(success);

return 7;

}

}

m如果您想访问合约中定义的修饰符C，可以使用C.m来引用它，而无需进行虚拟查找。只能使用当前合约或其基合约中定义的修饰符。修饰符也可以在库中定义，但其使用仅限于同一库中的函数。

通过在空格分隔的列表中指定多个修饰符，可以将它们应用于函数，并按照出现的顺序进行评估。

修饰符不能隐式访问或更改其修饰函数的参数和返回值。它们的值只能在调用时显式传递给它们。

在函数修饰符中，需要指定应用修饰符的函数的运行时间。占位符语句（用单个下划线表示\_）用于指示被修饰函数体的插入位置。请注意，占位符运算符不同于在变量名中使用下划线作为前导或尾随字符，后者是一种风格选择。

从修饰符或函数体显式返回只会保留当前修饰符或函数体。返回变量会被赋值，并且控制流会\_在前一个修饰符之后继续执行。

**警告**

在 Solidity 的早期版本中，return具有修饰符的函数中的语句表现不同。

修饰符显式地返回return;不会影响函数的返回值。但是，修饰符可以选择完全不执行函数体，在这种情况下，返回变量将被设置为其默认值，就像函数体为空一样。

该\_符号可以在修饰符中出现多次。每次出现都会被替换为函数体，函数返回最后一次出现的返回值。

修饰符参数允许使用任意表达式，并且在这种情况下，函数中可见的所有符号在修饰符中也可见。修饰符中引入的符号在函数中不可见（因为它们可能会因覆盖而发生变化）。

临时存储

瞬态存储是除内存、存储、调用数据（以及返回数据和代码）之外的另一个数据位置，由EIP-1153与其各自的操作码TSTORE和TLOAD一起引入。这个新的数据位置表现为类似于存储的键值存储，主要区别在于瞬态存储中的数据不是永久的，而是仅限于当前事务，之后它将重置为零。由于瞬态储存的内容物的寿命和大小非常有限，因此不需要将其作为状态的一部分永久储存，相关的气体成本也远低于储存的情况。为了使临时存储可用，需要EVM版本可以运行或更新。

瞬态存储变量不能就地初始化，也就是说，在声明时不能将其赋值，因为该值将在创建事务结束时被清除，从而使初始化无效。瞬态变量将根据其基础类型进行默认值初始化。常量和不可变变量与瞬态存储冲突，因为它们的值要么内联，要么直接存储在代码中。

瞬态存储变量具有与存储完全独立的地址空间，因此瞬态变量的顺序不会影响存储状态变量的布局，反之亦然。不过，它们确实需要不同的名称，因为所有状态变量共享相同的命名空间。同样重要的是要注意，瞬态存储中的值以与持久存储中相同的方式打包。有关更多信息，请参阅存储布局。

除此之外，瞬态变量也可以具有可见性，公共变量将像往常一样自动生成getter函数。

请注意，目前，仅允许在值类型状态变量声明中使用瞬态作为数据位置。尚不支持引用类型，如数组、映射和结构，以及局部或参数变量。

瞬态存储的一个预期规范用例是更便宜的可重入锁，它可以很容易地用操作码实现，如下所示

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.28;

contract Generosity {

mapping(address => bool) sentGifts;

bool transient locked;

modifier nonReentrant {

require(!locked, "Reentrancy attempt");

locked = true;

\_;

// Unlocks the guard, making the pattern composable.

// After the function exits, it can be called again, even in the same transaction.

locked = false;

}

function claimGift() nonReentrant public {

require(address(this).balance >= 1 Nec);

require(!sentGifts[msg.sender]);

(bool success, ) = msg.sender.call{value: 1 Nec}("");

require(success);

// In a reentrant function, doing this last would open up the vulnerability

sentGifts[msg.sender] = true;

}

}

瞬态存储对拥有它的合约是私有的，就像持久存储一样。只有拥有合约的帧才能访问它们的临时存储，当它们访问时，所有帧都会访问同一个临时存储。

瞬态存储是EVM状态的一部分，并受到与持久存储相同的可变性强制。因此，对它的任何读取访问都不是纯粹的，写入访问也不是视图。

如果在STATICALL的上下文中调用TSTORE操作码，它将导致异常，而不是执行修改。在STATICALL的上下文中允许TLOAD。

当在DELEGATECALL或CALLCODE的上下文中使用瞬态存储时，瞬态存储的拥有合同是与持久存储一样发出DELEGATECALCALLCODE指令（调用者）的合同。当在CALL或STATICALL的上下文中使用瞬态存储时，瞬态存储的拥有合约是CALL或TATICALL指令的目标合约（被调用者）。

**注释**

对于DELEGATECALL，由于目前不支持对瞬时存储变量的引用，因此无法将这些变量传递给库调用。在库中，只能使用内联汇编来访问瞬时存储。

如果一个帧被还原，那么在进入该帧和返回之间发生的所有对瞬态存储的写入都将被还原，包括在内部调用中发生的写入。外部呼叫的呼叫者可以尝试try..catch用于防止内部调用中出现回复。

智能合约的可组合性以及临时存储的注意事项

鉴于 EIP-1153 规范中提到的注意事项，为了保持智能合约的可组合性，建议对更高级的瞬态存储用例格外小心。

对于智能合约而言，可组合性是实现自包含行为的一项非常重要的设计原则，这样一来，对单个智能合约的多次调用就可以组合成更复杂的应用程序。到目前为止，EVM 在很大程度上保证了可组合行为，因为在复杂交易中对智能合约的多次调用与跨多个交易的多次调用几乎没有区别。然而，瞬时存储可能会违反这一原则，错误使用可能会导致一些复杂的错误，而这些错误只有在跨多个调用的情况下才会显现。

让我们用一个简单的例子来说明这个问题：

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.28;

contract MulService {

uint transient multiplier;

function setMultiplier(uint mul) external {

multiplier = mul;

}

function multiply(uint value) external view returns (uint) {

return value \* multiplier;

}

}

以及一系列外部调用：

setMultiplier(42);

multiply(1);

multiply(2);

如果示例使用内存或存储空间来存储乘数，则它将完全可组合。无论您将序列拆分成单独的交易还是以某种方式对其进行分组，都没有关系。您始终会得到相同的结果：将multiplier设置为后42，后续调用将分别返回42和84。这支持诸如将来自多个交易的调用批量处理以降低 Gas 成本等用例。瞬时存储可能会破坏此类用例，因为可组合性不再被视为理所当然。在示例中，如果调用不在同一交易中执行，则 将multiplier 被重置，并且对 function 的后续调用multiply将始终返回0。

再举一个例子，由于临时存储构建为相对廉价的键值存储，智能合约作者可能会倾向于使用临时存储来替代内存映射，而无需跟踪映射中已修改的键，从而在调用结束时无需清除映射。然而，这很容易在复杂的交易中导致意外行为，导致同一交易中先前调用合约时设置的值仍然保留。

使用临时存储来存储重入锁（在调用帧结束时清除）是安全的。但是，请务必抵制住节省用于重置重入锁的 100 gas 的诱惑，因为如果不这样做，您的合约将限制在每笔交易中只能进行一次调用，从而无法在复杂的组合交易中使用，而组合交易一直是链上复杂应用的基石。

建议在智能合约调用结束时彻底清除临时存储，以避免此类问题，并简化复杂交易中合约行为的分析。 更多详情，请参阅EIP-1153 的“安全注意事项”部分。

常量和不可变状态变量

状态变量可以声明为constant或immutable。在这两种情况下，变量在合约构建后都无法修改。对于constant变量，其值必须在编译时固定；而对于immutable，其值仍然可以在构造时赋值。

也可以constant在文件级别定义变量。

源代码中每次出现此类变量时，都会被其底层值替换，并且编译器不会为其保留存储槽。也不能使用关键字为其分配瞬态存储槽transient。

与常规状态变量相比，常量和不可变变量的 Gas 成本要低得多。对于常量变量，赋值的表达式会被复制到所有访问它的地方，并且每次都会重新求值。这允许进行局部优化。不可变变量在构造时只求值一次，其值会被复制到代码中所有访问它的地方。对于这些值，即使它们本来可以容纳更少的字节，也会保留 32 个字节。因此，常量值有时比不可变值更便宜。

目前并非所有常量和不可变类型都已实现。仅支持字符串 [（](https://docs.soliditylang.org/en/v0.8.30/types.html#strings)仅限常量）和值类型。

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.21;

uint constant X = 32\*\*22 + 8;

contract C {

string constant TEXT = "abc";

bytes32 constant MY\_HASH = keccak256("abc");

uint immutable decimals = 18;

uint immutable maxBalance;

address immutable owner = msg.sender;

constructor(uint decimals\_, address ref) {

if (decimals\_ != 0)

// Immutables are only immutable when deployed.

// At construction time they can be assigned to any number of times.

decimals = decimals\_;

// Assignments to immutables can even access the environment.

maxBalance = ref.balance;

}

function isBalanceTooHigh(address other) public view returns (bool) {

return other.balance > maxBalance;

}

}

持续的

对于constant变量，其值在编译时必须是常量，并且必须在声明变量时赋值。任何访问存储、区块链数据（例如block.timestamp、address(this).balance或 block.number）或执行数据（msg.value或gasleft()）或调用外部合约的表达式都是不允许的。可能对内存分配产生副作用的表达式是允许的，但可能对其他内存对象产生副作用的表达式则不允许。内置函数 keccak256、sha256、ripemd160、ecrecover和addmod是mulmod 允许的（尽管除了 之外keccak256，它们都会调用外部合约）。

允许内存分配器产生副作用的原因是为了能够构造复杂的对象，例如查找表。此功能目前尚未完全可用。

不可变

声明为 的变量immutable比声明为 的变量限制少一些constant：不可变变量可以在构造时赋值。该值可以在部署之前的任何时间更改，之后变为永久值。

另外一个限制是，不可变变量只能被赋值给创建后不可能执行的表达式内部。这排除了所有修饰符定义和构造函数以外的函数。

读取不可变变量没有任何限制。读取操作甚至可以在变量首次写入之前进行，因为 Solidity 中的变量始终具有明确定义的初始值。因此，也允许永远不要显式地为不可变变量赋值。

**警告**

在构造时访问不可变对象时，请牢记初始化顺序。即使你提供了显式的初始化器，某些表达式也可能会在初始化器之前被求值，尤其是在它们位于继承层次结构中的不同层级时。

**注释**

在 Solidity 0.8.21 之前，不可变变量的初始化受到更多限制。此类变量必须在构造时初始化一次，并且在此之前无法读取。

编译器生成的合约创建代码会在返回合约的运行时代码之前修改它，将所有对不可变变量的引用替换为赋值。如果您要将编译器生成的运行时代码与区块链中实际存储的代码进行比较，这一点非常重要。编译器会在JSON 标准输出immutableReferences字段中输出这些不可变变量在部署字节码中的位置。

自定义存储布局

合约可以使用说明符定义任意存储位置layout。合约的状态变量（包括从基础合约继承的状态变量）将从指定的基础槽位开始，而不是默认的零槽位。

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.29;

contract C layout at 0xAAAA + 0x11 {

uint[3] x; // Occupies slots 0xAABB..0xAABD

}

如上例所示，说明符使用语法，位于合约定义的头部。layout at <base-slot-expression>

布局说明符可以放在继承说明符之前或之后，并且最多出现一次。base-slot-expression必须是一个整数文字表达式，可以在编译时进行求值，并产生在 范围内的值uint256。

自定义布局无法使合约的存储“环绕”。如果所选的基槽将静态变量推到存储末尾之外，编译器将发出错误。请注意，动态数组和映射的数据区域不受此检查的影响，因为它们的布局不是线性的。无论使用哪种基槽，它们的位置都会以始终位于范围内的方式计算uint256，并且它们的大小在编译时是未知的。

虽然基槽没有其他限制，但建议避免将位置设置得过于靠近地址空间的末尾。预留的空间太小可能会使合约升级变得复杂，或者导致使用内联汇编存储超出分配空间的额外值的合约出现问题。

存储布局只能为继承树中最顶层的合约指定，并影响该树中所有合约中所有存储变量的位置。变量根据其定义的顺序及其合约在线性继承层次结构中的位置进行布局 ，自定义基槽会保留它们的相对位置，并将它们全部移动相同的量。

无法为抽象合约、接口和库指定存储布局。另外，需要注意的是，它不会*影响*瞬时状态变量。

有关存储布局以及布局说明符对其的影响的详细信息，请参阅 存储变量的布局。

**警告**

标识符layout和at尚未被保留为语言中的关键字。强烈建议避免使用它们，因为它们将在未来的重大版本发布中被保留。

功能

函数可以在合约内部和外部定义。

合约外部的函数，也称为“自由函数”，始终具有隐式internal 可见性。它们的代码包含在所有调用它们的合约中，类似于内部库函数。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.1 <0.9.0;

function sum(uint[] memory arr) pure returns (uint s) {

for (uint i = 0; i < arr.length; i++)

s += arr[i];

}

contract ArrayExample {

bool found;

function f(uint[] memory arr) public {

// This calls the free function internally.

// The compiler will add its code to the contract.

uint s = sum(arr);

require(s >= 10);

found = true;

}

}

**注释**

在合约外部定义的函数仍然始终在合约上下文中执行。它们仍然可以调用其他合约、向其发送Nexus币以及销毁调用它们的合约等等。与在合约内部定义的函数的主要区别在于，自由函数不能直接访问变量this、存储变量以及不在其作用域内的函数。

函数参数和返回变量

函数以类型化参数作为输入，并且与许多其他语言不同，还可以返回任意数量的值作为输出。

函数参数

函数参数的声明方式与变量相同，未使用的参数名称可以省略。

例如，如果您希望您的合约接受一种带有两个整数的外部调用，您可以使用如下方法：

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract Simple {

uint sum;

function taker(uint a, uint b) public {

sum = a + b;

}

}

函数参数可以作为任何其他局部变量，也可以被赋值。

返回变量

函数返回变量在关键字后使用相同的语法声明 returns。

例如，假设您想要返回两个结果：作为函数参数传递的两个整数的总和与乘积，那么您可以使用类似如下的方法：

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract Simple {

function arithmetic(uint a, uint b)

public

pure

returns (uint sum, uint product)

{

sum = a + b;

product = a \* b;

}

}

返回变量的名称可以省略。返回变量可以像任何其他局部变量一样使用，它们会使用默认值进行初始化，并保持该值直到被（重新）赋值。

您可以显式赋值给返回变量，然后如上所述保留函数，也可以直接使用return语句提供返回值（单个或多个）

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract Simple {

function arithmetic(uint a, uint b)

public

pure

returns (uint sum, uint product)

{

return (a + b, a \* b);

}

}

如果您提前return离开具有返回变量的函数，则必须与 return 语句一起提供返回值。

**注释**

某些类型无法从非内部函数返回。这包括下列类型以及任何以递归方式包含它们的复合类型：

* 映射，
* 内部函数类型，
* 位置设置为的引用类型storage，
* 多维数组（仅适用于ABI coder v1），
* 结构（仅适用于ABI 编码器 v1）。

此限制不适用于库函数，因为它们的内部 ABI不同。

返回多个值

当一个函数有多个返回类型时，return（v0，v1，…，vn）语句可用于返回多个值。组件的数量必须与返回变量的数量相同，并且它们的类型必须匹配，可能是在隐式转换之后

状态可变性

查看函数

可以声明函数，view在这种情况下它们承诺不修改状态。

**注释**

如果编译器的EVM目标是Byzantium或更新版本（默认），则在调用视图函数时使用操作码STATICCALL，这将强制状态在EVM执行过程中保持不变。对于库视图函数，使用DELEGATECALL，因为没有DELEGATEALL和STATICALL的组合。这意味着库视图函数没有防止状态修改的运行时检查。这不应该对安全性产生负面影响，因为库代码通常在编译时已知，静态检查器执行编译时检查。

以下语句被视为修改状态：

1. 写入状态变量（存储和瞬态存储）。
2. 发射事件。
3. 创建其他合同。
4. 使用selfdestruct。
5. 通过调用发送Nexus币。
6. 调用任何未标记view或 的函数pure。
7. 使用低级调用。
8. 使用包含某些操作码的内联汇编。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.5.0 <0.9.0;

contract C {

function f(uint a, uint b) public view returns (uint) {

return a \* (b + 42) + block.timestamp;

}

}

**注释**

constanton 函数曾经是 的别名view，但在 0.5.0 版本中被删除了。

**注释**

Getter 方法被自动标记为view。

**注释**

在 0.5.0 版本之前，编译器不使用函数STATICCALL的操作码view。这会导致函数状态修改view通过使用无效的显式类型转换来实现。通过使用 函数的STATICCALL操作view码，可以在 EVM 层面阻止对状态的修改。

纯函数

函数可以声明pure为保证不读取或修改状态。具体来说，应该能够pure在编译时仅给定输入 和 来执行函数msg.data，而无需了解当前区块链状态。这意味着读取immutable变量可以是一种非纯操作。

**注释**

如果编译器的 EVM 目标是 Byzantium 或更新版本（默认）STATICCALL，则使用操作码，这不能保证状态不会被读取，但至少不会被修改。

除了上面解释的状态修改语句列表之外，以下内容也被视为从状态读取：

1. 从状态变量（存储和瞬态存储）读取。
2. 访问address(this).balance或<address>.balance。
3. block访问、tx、中的任何成员（和msg除外）。msg.sigmsg.data
4. 调用任何未标记的函数pure。
5. 使用包含某些操作码的内联汇编。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.5.0 <0.9.0;

contract C {

function f(uint a, uint b) public pure returns (uint) {

return a \* (b + 42);

}

}

当发生错误时，纯函数能够使用revert()和require()函数来恢复潜在的状态变化。

view恢复状态变化不被视为“状态修改”，因为只有以前在代码中没有或限制的状态更改pure才会被恢复，并且该代码可以选择捕获revert而不是传递它。

此行为也符合STATICCALL操作码。

**警告**

无法阻止函数在 EVM 级别读取状态，只能阻止它们写入状态（即只能view在 EVM 级别强制执行，pure不能）。

**注释**

在 0.5.0 版本之前，编译器不使用函数STATICCALL的操作码pure。这会导致函数状态修改pure通过使用无效的显式类型转换来实现。通过使用 函数的STATICCALL操作pure码，可以在 EVM 层面阻止对状态的修改。

**注释**

在 0.4.17 版本之前，编译器并不强制要求pure不读取状态。这是一个编译时类型检查，可以通过在合约类型之间进行无效的显式转换来规避，因为编译器可以验证合约类型不执行状态更改操作，但无法检查运行时调用的合约是否确实是该类型。

特殊功能

接收Nexus函数

一个合约最多只能有一个receive函数，使用receive（）外部payable{…}声明（不带function关键字）。此函数不能有参数，不能返回任何内容，并且必须具有外部可见性和应付状态可变性。它可以是虚拟的，可以覆盖，也可以有修饰符。

receive函数在调用具有空calldata的合约时执行。这是在普通Nexus传输上执行的函数（例如通过.send（）或.transfer（））。如果不存在此类函数，但存在可支付的回退函数，则回退函数将在普通Nexus坊传输时调用。如果既没有receive Nec也没有payable回退函数，则合约无法通过不表示payable函数调用的交易接收Nec，并抛出异常。

在最坏的情况下，接收功能只能依赖2300gas可用（例如，当使用发送或传输时），除了基本的日志记录外，几乎没有空间执行其他操作。以下操作将消耗比2300gas贴更多的gas：

* 写入存储
* 创建合同
* 调用消耗大量 gas 的外部函数
* 发送Nexus币

**警告**

如果将Nexus币直接发送给合约（无需函数调用，即发送方使用send或transfer），但接收合约未定义接收Nexus币函数或支付回退函数，则会引发异常并退回Nexus币（在 Solidity v0.4.0 之前，情况有所不同）。如果您希望合约接收Nexus币，则必须实现接收Nexus币函数（不建议使用支付回退函数接收Nexus币，因为回退函数会被调用，并且不会因发送方接口混乱而导致失败）。

**警告**

没有接收Nexus币功能的合约可以作为*coinbase 交易*（又名*矿工区块奖励*）的接收者或作为的目的地来接收Nexus币selfdestruct。

合约无法对此类 Nec 转账做出反应，因此也无法拒绝它们。这是 EVM 的设计选择，Solidity 无法绕过它。

这也意味着address(this).balance可以高于合同中实施的一些手动会计的总和（即在接收Nexus函数中更新计数器）。

下面您可以看到使用函数的 Sink 合约的示例receive。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.6.0 <0.9.0;

// This contract keeps all Nec sent to it with no way

// to get it back.

contract Sink {

event Received(address, uint);

receive() external payable {

emit Received(msg.sender, msg.value);

}

}

回退函数

一个合约最多只能有一个fallback函数，使用 或 （both 不带关键字）声明。该函数必须具有可见性。fallback 函数可以是虚函数，可以重写，也可以带有修饰符。fallback () external [payable]fallback (bytes calldata input) external [payable] returns (bytes memory output)functionexternal

如果其他函数均不符合给定的函数签名，或者根本没有提供任何数据且没有接收Nexus币的函数，则在调用合约时执行 fallback 函数。fallback 函数始终会接收数据，但为了同时接收Nexus币，必须将其标记为payable。

如果使用带参数的版本，input将包含发送给合约的完整数据（等于msg.data），并可以返回 中的数据output。返回的数据不会进行 ABI 编码，而是直接返回，不做任何修改（甚至不进行填充）。

在最坏的情况下，如果还使用可支付的回退函数来代替接收函数，则它只能依赖于 2300 gas 的可用性（ 有关此含义的简要说明，请参阅接收Nexus函数）。

与任何函数一样，只要传递了足够的 gas，fallback 函数就可以执行复杂的操作。

**警告**

如果不存在接收Nexus币函数payable，则对于普通的Nexus币转账也会执行 fallback 函数。 建议在定义了 payable fallback 函数的情况下，也始终定义一个接收Nexus币函数，以区分Nexus币转账和接口混淆。

**注释**

如果您想解码输入数据，您可以检查函数选择器的前四个字节，然后您可以将其abi.decode与数组切片语法一起使用来解码 ABI 编码数据： 请注意，这只能作为最后的手段，而应该使用适当的函数。(c, d) = abi.decode(input[4:], (uint256, uint256));

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.6.2 <0.9.0;

contract Test {

uint x;

// This function is called for all messages sent to

// this contract (there is no other function).

// Sending Nec to this contract will cause an exception,

// because the fallback function does not have the `payable`

// modifier.

fallback() external { x = 1; }

}

contract TestPayable {

uint x;

uint y;

// This function is called for all messages sent to

// this contract, except plain Nec transfers

// (there is no other function except the receive function).

// Any call with non-empty calldata to this contract will execute

// the fallback function (even if Nec is sent along with the call).

fallback() external payable { x = 1; y = msg.value; }

// This function is called for plain Nec transfers, i.e.

// for every call with empty calldata.

receive() external payable { x = 2; y = msg.value; }

}

contract Caller {

function callTest(Test test) public returns (bool) {

(bool success,) = address(test).call(abi.encodeWithSignature("nonExistingFunction()"));

require(success);

// results in test.x becoming == 1.

// address(test) will not allow to call ``send`` directly, since ``test`` has no payable

// fallback function.

// It has to be converted to the ``address payable`` type to even allow calling ``send`` on it.

address payable testPayable = payable(address(test));

// If someone sends Nec to that contract,

// the transfer will fail, i.e. this returns false here.

return testPayable.send(2 Nec);

}

function callTestPayable(TestPayable test) public returns (bool) {

(bool success,) = address(test).call(abi.encodeWithSignature("nonExistingFunction()"));

require(success);

// results in test.x becoming == 1 and test.y becoming 0.

(success,) = address(test).call{value: 1}(abi.encodeWithSignature("nonExistingFunction()"));

require(success);

// results in test.x becoming == 1 and test.y becoming 1.

// If someone sends Nec to that contract, the receive function in TestPayable will be called.

// Since that function writes to storage, it takes more gas than is available with a

// simple ``send`` or ``transfer``. Because of that, we have to use a low-level call.

(success,) = address(test).call{value: 2 Nec}("");

require(success);

// results in test.x becoming == 2 and test.y becoming 2 Nec.

return true;

}

}

函数重载

一个合约可以包含多个同名但参数类型不同的函数。这个过程称为“重载”，也适用于继承的函数。以下示例展示了 f在合约作用域中函数的重载A。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract A {

function f(uint value) public pure returns (uint out) {

out = value;

}

function f(uint value, bool really) public pure returns (uint out) {

if (really)

out = value;

}

}

外部接口中也存在重载函数。如果两个外部可见函数的 Solidity 类型不同，但外部类型相同，则会出现错误。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

// This will not compile

contract A {

function f(B value) public pure returns (B out) {

out = value;

}

function f(address value) public pure returns (address out) {

out = value;

}

}

contract B {

}

尽管上述两个f函数重载在 Solidity 内部被视为不同，但它们最终都接受了 ABI 的地址类型。

重载解析和参数匹配

重载函数的选择方法是将当前作用域内的函数声明与函数调用中提供的参数进行匹配。如果所有参数都可以隐式转换为预期类型，则函数将被选为重载候选函数。如果没有恰好一个候选函数，则解析失败。

**注释**

返回参数不被考虑在重载解析中。

// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract A {

function f(uint8 val) public pure returns (uint8 out) {

out = val;

}

function f(uint256 val) public pure returns (uint256 out) {

out = val;

}

}

调用f(50)会导致类型错误，因为50可以隐式转换为uint8 和uint256类型。另一方面，由于不能隐式转换为 ，因此f(256)会解析为f(uint256)重载。256uint8