# 概述

交易有两种，在L2上发起的，和在L1上发起的优先交易，这跟OP是一样的。

整体流程：

1. 用户发起交易
2. Operator创建块，将交易加入块中
3. 块完成后，operator将其提交到L1的合约中，完成部分检查。
4. ZK证明被提交到L1，块完结

对交易来说，有4个状态：

* Pending：operator收到了交易，但是还没执行
* Processed：operator执行了交易，并且确认会加入到下一个块中
* Committed，已经被提交到了L1上，还没被证明
* Finalized，已经被ZK证明，获得了最终确定性。

当前一般从processed到finalized，时间在几个小时左右。

对开发来说，一般把processed和committed看作是同一个状态：committed。

# 当前状态

* 支持solidity 0.8.x，几乎不用改代码
* 除了少数例外，web3 API与以太坊兼容。
* 通过预编译，支持ecrecovery, keccat256, sha256
* 支持hardhat插件
* 支持L1->L2消息，允许开发者从L1向L2合约发消息。
* zkSync建议使用可升级合约，因为他们也担心zkEVM还有bug，用可升级合约保险点。

未来：zkPorter，应该是Validium版本。zkSync和zkPorter数据可以互通。

### 交易和块

从当前的区块浏览器来看，一个交易被接收后，立刻会出一个块，也就是一个交易一个块。然后很多块隔断时间会被一次性rollup到L1上。

根据文档，一个交易一个L2块是当时中心化的做法，以后L2去中心化后，就会变成几秒钟才出一个块了。

### L2的空块

当把L2打包提交到L1时，在被打包的L2交易之外，会有一笔将收集到的gas费转给矿工的交易，这笔交易在L2上表现为一个空块。

# 使用

现有的以太坊SDK都可以直接用。

如果要用zkSync的高级功能，比如账户抽象，就需要用zkSync SDK

合约部署需要用到zkSync，通过hardhat插件完成。

**Mainnet network info**

* Network Name: zkSync Era Mainnet
* RPC URL: https://mainnet.era.zksync.io
* Chain ID: 324
* Currency Symbol: ETH
* Block Explorer URL: https://explorer.zksync.io/
* WebSocket URL: wss://mainnet.era.zksync.io/ws

**Testnet network info**

* Network Name: zkSync Era Testnet
* RPC URL: https://testnet.era.zksync.dev
* Chain ID: 280
* Currency Symbol: ETH
* Block Explorer URL: https://goerli.explorer.zksync.io/
* WebSocket URL: wss://testnet.era.zksync.dev/ws

目前zkSync Era不支持EIP1559，可以在交易中指明{type:0}来使用以太坊老gas模型。

## SDK

zkSync有提供很多SDK，如web3j等，与以太坊上的用法一样，只不过返回的是L2的内容。如block.number,block.timestamp表示L2上面的。（但是合约里的block.number和block.timestamp代表L1上的）

当需要访问L1内容时，有新加一些接口。

### 与以太坊区别

<https://era.zksync.io/docs/dev/building-on-zksync/contracts/differences-with-ethereum.html#calldataload-calldatacopy>

大部分区别都是在opcode层面的。

#### 合约部署

与以太坊的部署有很大不同，需要调用系统合约[ContractDeployer](https://era.zksync.io/docs/dev/developer-guides/system-contracts.html#contractdeployer) 的create函数，参数中提供合约的hash和构造函数参数。合约的字节码需要在EIP712交易的factory\_deps 字段中提供。如果合约是工厂合约（可以部署其他合约），则其字节码需要在factory\_deps字段中引用（啥意思？跟提供有什么不同？）

Solidity：建议使用solidity 0.8.因为旧的可能有bug，并且可能编译器有局限。

VyPer：当前仅支持0.3.3

必须使用zkSync的编译器来编译合约：solidity用zksolc。

在zkSync里用了跟以太坊不同的地址推导方法。

用起来跟以太坊还是一样的，只是底层实现不一样：

MyContract a = new MyContract();

MyContract a = new MyContract{salt: ...}();

zkSync上的合约部署纯粹依赖字节码，它的合约地址生成规则跟以太坊不同，不过CREATE,CREATE2还是可以正常使用的。对CREATE来说，生成地址分为2部分：deploymentNonce和transactionNonce，deploymentNonce是该账户用CREATE部署合约的次数，transactionNonce是防止重放攻击的（跟以太坊nonce一样？）。比较安全的方法是监听ContractDeployed事件，获得新合约的地址。CREATE2也是支持的，是ContractDeployer合约的接口，可以获得固定的地址，另外也是EOA可调用的。

也可以通过L1->L2的有限交易来部署合约，这跟其他的优先交易是一样的。区别就是，合约字节码不用再rollup到L1了，因为已经放过了。

合约的部署是通过前端系统和ContractDeployer系统合约配合完成的。系统合约中传入合约的hash而不是字节码，合约字节码通过EIP712的字段传入。

合约的calldata包含如下内容：

1. The signature (4 bytes).
2. The salt (32 bytes).
3. The contract hash (32 bytes).
4. The constructor calldata offset (32 bytes).
5. The constructor calldata length (32 bytes).
6. The constructor calldata itself (N bytes).

1-5取代原先的合约字节码，6还是构造函数参数。所以当调用CREATE或CREATE2时，会传送132+N byte的calldata到ContractDeployer，然后返回新合约地址。如果同时有传输ETH币，则ContractDeployer会调用MsgValueSimulator来传输币。

建议用new而不是assembly来创建合约，但如果真用了assembly，那就得核对看看是否符合以上规则。

zkSync里不存储合约的字节码，而是特殊格式的hash，字节码保存在transaction数据中。一旦transaction被成功执行，字节码会被发布到L1上。

如果你的合约是工厂合约（会部署其他合约），工厂合约并不会保存它要部署的其他合约的字节码，而是保存字节码的hash，要在factory\_deps里包含所有字节码。

注意：factory\_deps里的内容并不是交易执行所必需的。它们只是被标记需要放到L1上。所以如果你的factory\_deps里包含太多内容，以至于一个L1交易塞不下，你可以把它放在多个L1交易里。例如如果你要部署合约A，A会再部署B,C，所以A,B,C的字节码都需要放在factory\_deps里，放到L1上。假设A,B,C合起来太大。你可以发一个空交易，交易的factory\_deps里放A,B的字节码（假设能塞下），并且执行合约部署交易；然后再发第二个交易，第二个交易的factory\_deps里包含C的字节码。但需注意，如果合约太大，一个交易里放不下，那这个合约需要被拆成更小的多个合约。

字节码格式：

* 长度（bytes）必须能被32整除
* 长度必须是单数，即 bytecode\_length % 64 = 32
* 不大于2^16 32bytes，即不大于2^21 bytes

字节码Hash格式：

* 前两个字节代表hash的格式，当前是[1,0]
* 随后的2个字节代表字节码长度（以32bytes计算）
* 后续的28个字节是sha256(bytecodes)的后28字节。

整个合约部署部分还是不太懂。

#### CALL， DELEGATECALL, STATICCALL

1. 返回值的内存处理差异，以太坊的返回值内存是在调用前分配的，而zkSync是在调用结束后，按需分配。
2. 在以太坊中，可以通过CALL{value:xxx}()来通过跨合约调用同时转账。  
   在zkSync里，没有原生支持以太坊转账，而是用一个模拟的系统合约：MsgValueSimulator来处理，它接收callee的地址，以太数量，进行检查和余额调整后，调用callee。

#### MSTORE，MLOAD

1. EVM里，内存增长是按字的，而在zkSync里，是按字节的。例如mstore(100,0)，在zkevm里，msize是132，但在evm里，是160.
2. Gas费用不同。EVM里是平方增长的，zkSync里是线性增长的。
3. zkSync对内存的使用有优化，所以有时候会使用更少的内存。

#### CODESIZE

在Yul里，用datasize来区分运行时的codesize和部署时的构造函数和参数size，但是在zkSync里，datasize=0，用codesize和calldatasize来表明上面那两个。

#### CODECOPY

略

#### RETURN

在构造函数中，Constructors return the array of immutable values. If you use RETURN in an assembly block in the constructor in zkSync Era, it will return the array of immutable values initialized so far.

#### Block.number和block.timestamp

在SDK里（如zkSync的web3j）返回的block.number和block.timestamp，对应L2上的block。

在合约（EVM）里的block.number和block.timestamp，对应L1 batch上的。

#### Revert block

有个revertBlock的接口，如果一个交易有问题，在它被提交到L1前，是可以revert的。但是revert的权限目前还不确定是否普通用户也有。

#### Hash

L2上的块hash计算方法：keccak256(l2\_block\_number)，所以它是确定性的，注意不要用L2的块hash作为随机数源。

#### 使用非inline的库

使用inline的库跟普通合约开发类似，但使用非inline的库（即通过合约地址调用）时，需要在config文件里配置一下。

在harhdat.config.ts里，在zksolc里添加libraries,例如：

import "@matterlabs/hardhat-zksync-deploy";

import "@matterlabs/hardhat-zksync-solc";

module.exports = {

zksolc: {

version: "1.3.10",

compilerSource: "binary",

settings: {

libraries: {

"contracts/MiniMath.sol": {

MiniMath: "0xF9702469Dfb84A9aC171E284F71615bd3D3f1EdC",

},

},

},

},

defaultNetwork: "zkTestnet",

networks: {

zkTestnet: {

url: "https://testnet.era.zksync.dev", // URL of the zkSync network RPC

ethNetwork: "goerli", // Can also be the RPC URL of the Ethereum network (e.g. `https://goerli.infura.io/v3/<API\_KEY>`)

zksync: true,

},

},

solidity: {

version: "0.8.13",

},

};

#### CoinBase

总是返回BootLoader合约的地址，即0x8001

#### Difficulty

总是返回2500000000000000

#### BASEFEE

zkSync当前不支持EIP1559.大部分情况下这个值是0.25gwei，在L1的gas费很高时会涨。

#### SELFDESTRUCT

被认为是有害的，所以弃用了。总是返回一个预定义错误。

#### CALLCODE

弃用了，总是返回预定义错误

#### PC

不支持，总返回预定义错误

#### EXTCODECOPY

合约字节码在当前架构下是无法获取的，只能获取CODESIZE和EXTCODESIZE

总是返回预定义错误

#### Immutable

因为zkEVM无法读取合约字节码，所以immutable是用系统合约模拟的。

在构造函数里，immutable被放在一个辅助heap里，被分配给index和值。他们被传给一个系统合约ImmutableSimulator，该系统合约用mapping保存这些immutables，key是新合约地址。

当使用的时候，新合约会调用ImmutableSimulator，获取immutable的值。

#### SDK

与前端交互时，需要用zksync的库，而不是原先的web3j, ethers。

//import { utils, Provider, Contract, Wallet } from "ethers";

import { utils, Provider, Contract, Wallet } from "zksync-web3";

## 可升级合约

使用hardhat-zksync-upgradable插件。这个插件是在openzeppelin的库上改的。

还在alpha测试，支持透明升级，不支持UUPS升级。可升级proxy和beacon proxy。

不支持升级校验。