UFC-chain

UFC Chain utilizes a decentralized model to complete cross chain digital assets transfer without relying on centralized exchanges. Yet, at the same time there is no need to code.

Through a simple purchasing model in conjunction with a flexible smart contract. UFC Chain can complete the cross-chain transfer of digital assets. Also, realize various asset transaction functions on the chain.

Cross Chain

Through cross-chain technology, we are able to transition digital assets on various chains to our UFC Chain for trading. The cross-chain function has the following benefits :

Alleviate the transaction congestion problem of mainstream digital assets

Solved the problem of non-mainstream digital asset liquidity issue

Addresses the need for direct rigid exchange of multiple assets

Solved the problem of large price fluctuations in large transactions

Smart Contract

By using our smart contracts, you are able to flexibly expand and customize complex transaction logic and complex financial contracts. At the same time, on the basis of simplifying and providing a code environment, the function can be dynamically expanded with restrictions and control without the need of adding new code. Every time a smart contract on the chain is called for execution, it will first initialize an independent lightweight execution environment in order to find the contract bytecode on the chain, then execute the contract bytecode. During execution, the on-chain data can be accessed through the native API . The chain provides native APIs for common operations so that smart contracts can have better performance in general.

Each smart contract has its own independent area, called storage. Only the change of contract storage is recorded on the chain. The amount of contract storage will automatically determine whether the contract result is on the chain.

2.1 Contracts and virtual machines

Our chain utilizes a Turing complete and custom-designed bytecode specification for our smart contract's virtual machine implementation specification. Compilers that provide high-level programming languages, such as C #, Java, TypeScript, etc., are able to generate smart contract bytecode from these programming languages.

Smart contract virtual machine:

The smart contract virtual machine is implemented as a Turing complete bytecode virtual machine.

Smart contract language:

Use a subset of the main features of existing programming languages such as C#Java, TypeScript and other popular programming languages as a priority programming language for smart contracts, compile to bytecode that conforms to the smart contract bytecode specification, to construct our smart contract.

Built-in library for smart contracts:

Smart contracts provide some basic libraries for commonly used numerical operations, string operations, etc., as well as some built-in function libraries for on-chain queries, transactions, etc. The built-in libraries can be called in smart contracts.

Smart Contracts Usage:

After our smart contracts are deployed, users are able to directly utilize their digital assets or save their digital assets onto the smart contracts. In addition, users are able to utilize other smart contracts to expand the functionality of the blockchain.

Part of the functional logic can be implemented as a smart contract and deployed on the chain. As a third-party library, it is used by other smart contracts on the chain to expand the function of the blockchain.

Functional scope and limitations of smart contracts:

Smart contracts use Turing complete programming language to do the following things. It is able to query data on the chain and access the storage of this contract. Also, it utilizes other smart contracts/native contracts. Lastly, it provides return information to users.

Limitations： Unable to attend data from other blockchain； Unable to confirm on the logic of each individual notes ； The number of instructions executed and the amount of memory space used are controlled by the blockchain; The blockchain can immediately terminate the execution of smart contracts at any time, such as when the contract execution cost exceeds the budget.

Smart contract state storage:

Each smart contract has an independent state space, called storage. Storage format is an unstructured data structure. The storage of smart contracts on the chain changes, rather than storing the latest storage on the chain every time. For example, in a contract call, the contract storage is changed from {"name": "chain"} to {"name": "chain", "count": 123}, and only the changed part is recorded on the chain {"count": 123}, and even when the contract invoice is charged, the storage part charges only calculate the size of the changed instead of the size of the total storage. Thus, even if a smart contract's state storage space is large, as long as the amount of change generated by each call to the contract is not large, the chain 's data increment and handling fee are not high.

Smart contract status query:

The smart contract can directly query part of the storage value of this contract, and can also retrieve part of the data in the nested data structure through the SQL-like programming language. When the storage of the smart contract is large, you can reduce the data load in this way to increase the query speed, avoid full table scans, and increase the performance upper limit of the data access part of the smart contract.

For example: the storage structure of a smart contract is similar

```
{
 “name”: “blockchain”,
 “userBalances”: [
   { “userAddress”: “a”, “amount”: 10000, “freeze”: false },
   { “userAddress”: “b”, “amount”: 20000, “freeze”: true },
   { “userAddress”: “c”, “amount”: 30000, “freeze”: false },
     ....... More data, Example : few hundred thousand lines of codes.
 ]
}
```

Users can use a SQL-like syntax like var frozendUsers = storage.query (“select userBalance.userAddress from userBalances as userBalance where freeze = true”) to query all the user addresses of the account frozen in this smart contract. Which will reduce the amount of data read and written by avoiding the full table scan. Thus, meeting the business scenarios need where more data is stored in the smart contracts. But the amount of reading is not applicable. Examples for these are simple push exchange in smart contracts, smart contract assets, contract insurance, etc. .

Smart contract life cycle:

Generate smart contract bytecode files through popular programming languages or manually constructing bytecodes

Deploy the smart contract bytecode to the blockchain, which can be created as a smart contract or as a smart contract template for use in the next contract creation

Utilize the smart contract API or transfer funds to the smart contract address

Each time the blockchain calls a smart contract, it first initializes an independent lightweight smart contract sandbox execution environment, loads and executes the smart contract in it

After executing the smart contract, according to whether the execution exit status is abnormal or not, save the execution result and contract storage changes

2.2 Consensus random number generator

The contract has the need to obtain a consensus random number. In order to generate a consensus random number, the input must be chain related data. There are two methods for obtaining random numbers:

Simple random number: directly call an interface in the contract to obtain a random number and provide a random number based on the current random seed

Complex random number: The user specifies a set of consecutive blocks in the contract. The system takes the prev_secret of this set of blocks as input to generate a random number. The user can designate a set of ungenerated blocks to be recorded in the contract. After the block is generated, the random number is determined.

The user can directly call the interface in the contract to obtain a simple random number. By this way, when the stakeholder of the execution result happens to be the current block producer, there is a possibility that the block producer chooses not to pack the call according to the random number result for its own interests.

In order to avoid that scenario, we can utilize the complex random numbers. The complex random number takes the Continuous Blocks prev_secret as input. If the block producer wishes to generate a random number that is useful to himself, he needs to modify the prev_secret of the current block according to the prev_secret of other blocks in the chain, but prev_secret is in the preceding block round It has been decided that it can not be adjusted, that is, the block producer can not regulate the generation of random numbers.

Consensus algorithm（UPoS）

UFC Chain uses decentralized consensus algorithm UPoS (Union Statement of Stake). The user can register as a Miner candidate, and can provide this Miner candidate with appropriate assets pledged to become a Miner by himself or other users. 25 Miners are selected according to the pseudo-random numbers on the chain at the beginning of each round of block generation, and then they are packed into order. Every round of block node elections will be chosen based on the weight of the pledge deposit made by Miner. In a round of block generation, the Miner with more deposits is more likely to be selected as a block node. After a Miner generates a block, Miner and Miner supporters can get a block reward based on the proportion of the pledged assets, including a new block reward, UFC handling fee, and partial side chain asset handling fee.

In addition to the other rights, Miner also has the right to review the Validator proposal if Miner does not complete the processing of the proposal within the deadline. It will affect Miner's participation rate. The participation rate will participate in the weighted calculation of Miner's bookkeeping rights.

There are maximum of 15 Validator users on the chain, and the Validator users jointly multi-sign and manage the side chain assets on the chain. The election and withdrawal of Validator is first nominated by the existing Validator to create a candidate proposal to modify the existing Validators. After reaching a consensus of not less than 2/3 between the existing Validator, it is converted into a proposal to modify the Validator. The final decision is based on the weight of the miners' deposit. The larger the deposit, the more power it has. Think of it like a company. When a proposal to modify Validator gets a vote of not less than 2/3, the proposal passes and the Validator changes. When the Validator changes, the multi-signature hot and cold wallet replacement process of the side chain will start at the same time.

The UPoS algorithm allows a block to be generated every 8 seconds, and only one authorized producer can generate blocks at any point in time. If a block is not produced within the specified time, the producer of this block will be skipped, and the next Miner account will be used to replace the block. When one or more Miners fail to generate blocks, there will be a delay of 10 seconds or more on the blockchain.