EQcoin Bible

Xun Wang

https://t.me/EQcoinUinverse

www.eqcoin.org

Table of contents

1. Terms

1. EQcoin is the original commodity of EQcoin ecosystem. EQcoin is a cryptocurrency, hereinafter referred to as "**EQC**".
2. "**ID**" is an abbreviation for "identity".
3. Type represents the type of lock, hereinafter referred to as "**T**".
4. Operation is defined in Section 2.7.1 below, hereinafter referred to as "**OP**".
5. Elliptical curve cryptography, hereinafter referred to as "**ECC**".
6. "**nonce**" is an abbreviation for "number used only once".
7. Transactions per second, hereinafter referred to as "**TPS**".
8. Proof of Stake, hereinafter referred to as "**PoS**".
9. Proof of Work, hereinafter referred to as "**PoW**".

2. EQcoin Overview

2.1 About EQcoin

EQcoin is the world's first Passport-based Decentralized Finance (DeFi) ecosystem of the people, by the people, for the people. EQcoin is the next-generation blockchain representing the ultimate game-changer. EQcoin is an open-source, decentralized, permissionless, distributed, and publicly shared digital ledger. Passport and EQC (an abbreviation for "EQcoin") are the original commodities of the EQcoin ecosystem. Passport and EQC are issued and circulated in accordance with the EQcoin consensus mechanism, which operates on decentralized finance principles. This enables everyone to participate in the issuance and circulation of Passport and EQC through crowdsourcing. The development of EQcoin is based on crowdsourcing. Everyone can contribute to the enhancement and refinement of EQcoin through the EQcoin Improvement Proposal (EIP).

2.1.1 Features of EQcoin

2.1.1.1 Passport-based blockchain module

The founder of EQcoin, Xun Wang, invented the world's first Passport-based blockchain model. Xun Wang owns all related intellectual property rights and copyrights, which are protected by relevant laws and regulations, such as ownership, intellectual property rights, and copyrights. Unauthorized use and quotation are strictly prohibited, and any infringement will be prosecuted by law. Passport is the cornerstone of the EQcoin ecosystem. Passport is user-controlled can be used for depositing digital assets, sending transactions, and deploying an independent smart contract. The Passport has a Status, an ID, a Balance, a Nonce, a LockNonce, a Lock, a PublicKey, and a SmartContract. Each passport is assigned a unique natural number identifier when it is issued, which serves as its ID. Passport can be referenced using its ID when sending transactions. The ID numbering starts from 1 and increases sequentially based on the order of Passport issuance. It enables users to easily manage and trade digital assets, enhancing their efficiency, trust, comfort, satisfaction, and loyalty by reducing complexity and providing an intuitive, user-friendly experience akin to that of an iPhone. Passport can not only send transactions by referring to the Passport ID related to the transaction but also provide

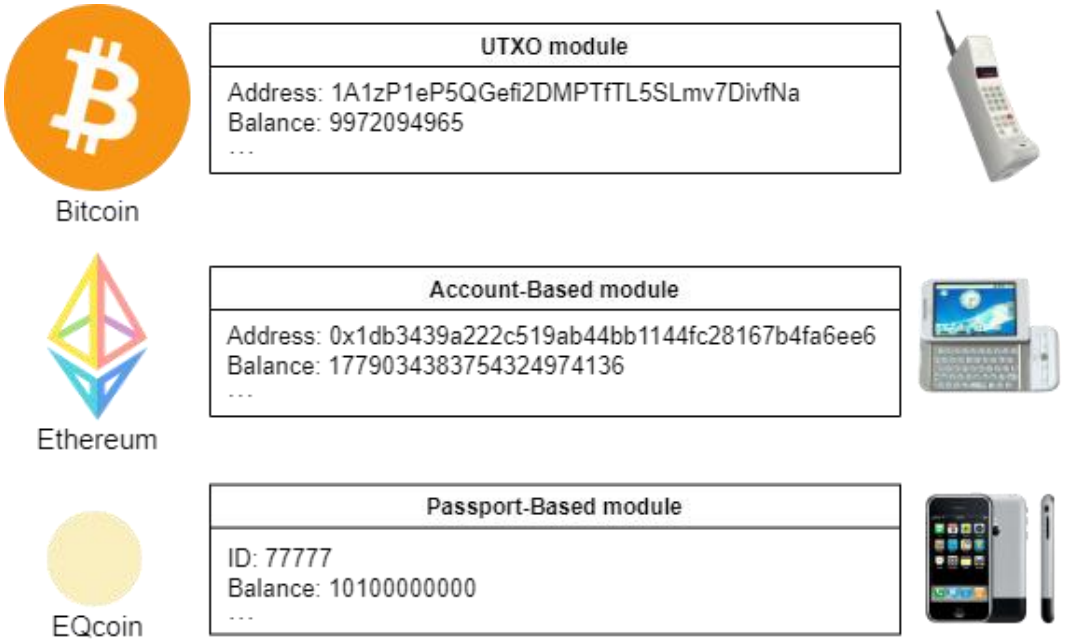
independent smart contract deployment services. This gives it a value similar to premium phone numbers, unique QQ numbers, domain names, ENS, and ENS domain names.

2.1.1.1.1 Passport samples

Adam's Passport
Status: 0
ID: 1
Nonce: 0
Balance: 2100000000000000
Lock: 0xaa...aa
PublicKey: null

Eve's Passport
Status: 0
ID: 2
Nonce: 0
Balance: 10100000000
Lock: 0xbb...bb
PublicKey: null

2.1.1.1.1 Blockchain Model Comparison of Bitcoin, Ethereum, and EQcoin



2.1.1.2 Supports performing multiple different operations in the same transaction

Users can perform multiple operations such as Transfer, SmartContract, IssuePassport, ChangeLock, and multiple OPs at the same time in the same transaction, thus saving transaction fees and providing a better user experience.

2.1.1.3 User-based issuance and sale of Passports and deployment of smart contract services.

Passport owners can provide services for issuing and selling passports, as well as deploying smart contracts for all users.

2.1.1.4 A currency supply model with a constant total supply and is much more scarce than Ethereum

EQcoin effectively combats inflation by implementing a currency supply model with a constant total supply, similar to Bitcoin. The decimal of EQcoin is 8, and its circulation and annual issuance are lower than those of Ethereum, which makes it more scarce and more resistant to inflation than Ethereum.

2.1.1.4.1 EQcoin and Ethereum currency supply comparison

Compare items	EQcoin	Ethereum
Genesis block supply	21,000,000~70,000,000 ETH	72,000,000 EQC
Annual supply	6,307,200 EQC	About 6,400,000 ETH
Decimal	8	18

Constant total supply	Yes	No

Note1: Ethereum currency supply data comes from Conducting the ETH Census - by Kyle Waters (substack.com).

Note2: According to the above data sources, the current total issuance of Ethereum is about $1.2*10^8$ ETH = $1.2*10^{28}$ wei. The maximum supply of EQcoin is $2.1*10^{11}$ EQC = $2.1*10^{19}$ singularity. Comparing their smallest denomination, the current total issuance of Ethereum is approximately $(1.2*10^{28})/(2.1*10^{19})$ = 571,428,571 times the maximum supply of EQcoin. So EQcoin is a much more scarce resource than Ethereum.

Note3: To sum up, with the continuous growth of EQcoin users and the ongoing development of the ecosystem, as the network effect of EQcoin accumulates to a certain extent, the quantitative change will lead to a qualitative change. Consequently, its price is expected to surpass that of Ethereum sooner or later.

2.1.1.5 Support the deployment and operation of more cost-effective smart contracts compatible with Ethereum

EQcoin supports the deployment and execution of more cost-effective smart contracts that are compatible with Ethereum, establishing a decentralized finance ecosystem that is more cost-effective than Ethereum.

2.1.1.6 High TPS and low transaction fees

EQcoin achieves a high Transactions Per Second (TPS) comparable to EOS and low transaction fees comparable to Polygon. This provides users with a faster and more cost-effective transaction experience compared to Ethereum.

2.1.1.7 Flexibility to combine and extend protocols and state objects

EQcoin enables the flexibility to combine and extend protocols and state objects, facilitating adaptation to evolving requirements by modifying protocol and state object statuses. Additionally, it can meet new demands by extending protocols and state objects.

2.1.1.8 Minimization of state data

EQcoin significantly reduces the cost of operating a full node by implementing state data minimization. This enables more users to afford the operational costs and ensures the decentralization of EQcoin.

2.1.1.9 Consistency in transaction sorting and execution order

EQcoin ensures consistency in transaction sorting and execution order, effectively preventing fraud and meeting the relevant requirements of decentralized applications (DApps), such as decentralized exchanges (DEX) or auctions.

2.2 About Passport

Passport is the cornerstone of the EQcoin ecosystem. Passport is user-controlled can be used for depositing digital assets, sending transactions, and deploying an independent smart contract. The Passport has a Status, an ID, a Balance, a Nonce, a LockNonce, a Lock, a PublicKey, and a SmartContract. Passport owners can provide services for issuing and selling passports, as well as deploying smart contracts for all users. They have the authority to determine the amount of EQC to be

charged for passport issuance, sales, and deployment of smart contracts. Just like a Bitcoin address and an Ethereum account, a Passport is anonymous and does not contain personal information about the owner.

2.2.1 PoS based Passport issuance consensus mechanism

In order to issue a new Passport, a specific amount of EQC must be staked in it.

2.2.1.1 List of the amount of EQC that needs to be staked for different Passports

Passport ID No.	The amount of EQC that needs to be staked
0~9	51
10~99	51
100~999	51
1000~9999	51
10000~99999	51
100000~999999	51
1000000~9999999	51

10000000~99999999	25.5
100000000~999999999	12.75

2.2.2 About Status

Status uses the bit flags to record the current Passport-related status, such as whether there is a smart contract deployed, staking mode, whether it is active, etc.

The type of Status is EQCBits. Using the Status, Passport can add or delete Passport relevant state objects created by the EQcoin Improvement Proposal to add or delete its specific functions.

2.2.3 About ID

Each Passport is assigned a unique natural number identifier when it is issued, which is its ID. Passport can be referenced using its ID when sending transactions. The ID numbering starts from 1 and increases sequentially based on the order of Passport issuance.

2.2.4 About Balance

Balance stores the balance of the Passport in Singularity.

2.2.5 About Nonce

Nonce of a Passport records the total number of transactions that have been sent and confirmed by this Passport using a specific Lock. Nonce ensures that the transaction signed by the Lock bound to the current Passport is unique to the current Lock, thereby avoiding double-spending attacks.

The value starts at 0 and increases by 1 with each sent and confirmed transaction, and is reset to 0 after each Passport Lock change.

2.2.5 About LockNonce

LockNonce of a Passport records the total number of times the current Passport has changed its Lock. LockNonce itself is not a component of the transaction. When signing and verifying transactions, LockNonce will be superimposed as dependency-injected data behind the original data of the transaction and treated as a whole for signing and verification.

The value starts at 0 and increases by 1 after each Passport lock change.

2.2.6 About Lock

Lock is used to safeguard and control the ownership of the Passport, ensuring that only its owner can use it to send transactions, thus preventing it from being illegally stolen by others. Locks are implemented based on various encryption technologies such as Elliptical curve cryptography (ECC) and SHA-3. Lock has a Type, Body and possibly other relevant state objects. Lock consists of two lock types at the current stage. New locks can be created through EQcoin Improvement Proposal to extend the functionality of the lock.

2.2.6.1 About Type

Type represents the type of Lock. The type of Type is EQCBits.

2.2.6.2 About Body

Lock body is implemented based on various encryption technologies. It is the main part of the lock. Different types of locks may have different lock body implementation methods.

2.2.6.3 Elliptical curve cryptography (ECC) based Lock

Elliptical curve cryptography (ECC) based Lock use ECC technology to safeguard and control the ownership of relevant

Passports.

2.2.6.3.1 T0 lock

T0 lock's lock type is 0 and use secp256r1(NIST P-256) elliptic curve to safeguard and control the ownership of relevant Passports. T0 lock has a lock type 0, a SHA3-256 public key hash of secp256r1(NIST P-256) elliptic curve. It has self-check-based error correction capabilities that can detect its character errors caused by reading, input or network transmission, thereby ensuring its correctness.

2.2.6.3.2 T1 lock

T1 lock's lock type is 1. It can provides multi-party safeguard and control the ownership of relevant Passport and 0-trust-based Passport security control services against loss and damage of private keys. T1 lock has a LockType, a Status, a SHA3-256 hash of the relevant redundant lock pairs. It has self-check-based error correction capabilities that can detect its character errors caused by reading, input or network transmission, thereby ensuring its correctness.

The T1 lock supports the following functions:

1. User is allowed to create a redundant lock pair, which contains N (1<=N<=4) T0 locks from different devices provided by the user. User can select a lock to unlock from the redundant lock pair. Thus, if a single device is damaged, the locks in the redundant devices are still available for use.
2. Single user is allowed to create N(1<=N<=8) lock pairs, and select the M(1<=M<=N) locks in the N lock pairs to unlock.
3. N(1<=N<=8) users are allowed to provide one lock pair per user and must use the N locks in the N lock pairs to unlock.

2.2.6 About PublicKey

PublicKey has a Status, one or more public keys that used to verify Passport relevant digital signatures. Because the corresponding public keys is stored in relevant Passport when the lock is unlocked for the first time, so that it can resist preimage attacks.

2.2.7 SmartContract

2.2.7.1 About smart contracts

Smart contracts are similar to the contracts and agreements in the real world. The only distinction is that they are digital. In fact, a SmartContract is a specialized type of computer

program designed to execute, control, or document events and actions in accordance with the terms of a contract or an agreement. It has a Status, an Identity, a Balance, a Nonce, a CodeHash and a StateRoot. It runs on EQCVM, itself and its related state object data are stored on the EQcoin blockchain. Once this data is recorded, it becomes traceable and irreversible.

Smart contracts are written in Intelligent. Smart contracts enable trusted transactions, contracts, and agreements to be conducted among diverse, anonymous parties without requiring a central authority, legal system, or external enforcement mechanism.

Smart contract is deployed in Passport. Each Passport can deploy an independent smart contract and it has the same ID as the Passport. Smart contract can be referenced using its ID.

2.2.7.2 About SmartContract

SmartContract stores the state objects related to the smart contract currently deployed in Passport.

2.2.7.2.1 About Status

Status uses the bit flags to record the current SmartContract

-related status.

The type of Status is EQCBits. Using the Status, SmartContract can add or delete SmartContract relevant state objects created by the EQcoin Improvement Proposal to add or delete its specific functions.

2.2.7.2.2 About Balance

Balance stores the balance of the SmartContract in Singularity.

2.2.7.2.3 About Nonce

Nonce of a SmartContract records the total number of transactions that have been sent and confirmed by this SmartContract. The value starts at 0 and increases by 1 with each sent and confirmed transaction.

2.2.7.2.4 About CodeHash

CodeHash stores the hash of the smart contract code.

2.2.7.2.5 About StateRoot

StateRoot stores the state objects' MerklePatriciaTrie root of the smart contract currently deployed in Passport.

2.3 About EQC

EQC is an abbreviation for "EQcoin". It is the original commodity of EQcoin ecosystem. It is a cryptocurrency. Users need to pay EQC to use EQcoin decentralized financial services.

2.3.1 About Singularity

Singularity is the smallest denomination of EQC. One EQC = 100,000,000 singularity (10^8).

2.4 Passport and EQC total supply

The max total supply of Passport is 4,294,967,296.

The total supply of EQC is a constant 210,000,000,000 and the decimal is 8. The first block, the Singularity block, will issue at least 21,000,000 EQC but no more than 70,000,000 EQC, and then will issue 6,307,200 EQC every year.

2.5 About Intelligent

Intelligent is an object-oriented programming language designed for developing smart contracts that run on EQcoinVM. It is compatible with Solidity. It is statically typed, supports inheritance, an Intelligent Standard Library, and

complex user-defined programming.

2.5.1 About Intelligent Standard Library (ISL)

The Intelligent Standard Library (ISL) provides a wide range of features that significantly expand the core Intelligent language, enhancing its versatility. The ISL is a collection of algorithms, data structures, and other components that can be used to simplify the development of Intelligent programs. The ISL is a collection of Intelligent contracts and interfaces designed to provide commonly used programming data structures and functions, such as tokens, utilities, access control, upgrades, etc. One of the primary advantages of ISL is that it provides the ability to import specific versions of contracts and interfaces included in ISL through an import statement, so there is no need to include ISL-related source code when deploying smart contracts. This can significantly reduce the cost of deploying smart contracts.

2.6 About EQcoin virtual machine(EQcoinVM)

EQcoin virtual machine(EQcoinVM) is a crucial component of the EQcoin blockchain. It serves as the runtime environment for managing the state of state variables , enabling smart

contract functionality, executing smart contracts and decentralized applications (DApps). It operates as a decentralized computer that runs on the global network of EQcoin nodes. It is responsible for processing and executing code written in EQcoin's native programming language, Intelligent. It is a Turing-complete, sandboxed execution environment. It is compatible with the Ethereum Virtual Machine.

2.7 About EQswap

EQswap is a decentralized exchange that uses an order book system to facilitate the trading of digital assets on the EQcoin blockchain. As a decentralized exchange, EQswap is permissionless, allowing everyone to trade digital assets or create a new market for exchanging a new pair of digital assets.

2.8 About Transaction

Transaction is essentially a signed set of instructions from one Passport. Transaction is used to affect a state change on the EQcoin blockchain, such as transfer of digital assets, change the lock of Passport, deploy smart contract, or execute

a function within a smart contract.

Transaction has a Status, a Passport ID, a Nonce , one or more body arrays, and a Signature. Using the Status, Transaction can add or delete bodies created by the EQcoin Improvement Proposal to add or delete its specific functions.

2.8.1 About Transaction Nonce

Transaction Nonce is a number that is used only once in a specific transaction. EQcoin network stipulates that the nonce of a new transaction must be the current Nonce of its Passport plus one. Therefore, each new Transaction Nonce must be set to the current Nonce of its Passport plus one.

Transaction Nonce enables preventing replay attacks, which involve a malicious user trying to resend a previous transaction. It ensures that each transaction can be uniquely identified, ordered correctly, processed and verified accurately within the EQcoin blockchain.

2.8.2 About Operation

Operation is essentially a set of instructions from one Passport. Operation is used to affect a state change on the EQcoin blockchain, such as change Passport's lock or deploy a

smart contract. Each OperationBody contains one or more operations.

Operation has an OP ID and one or more OP state objects. New operations can be created through EQcoin Improvement Proposal to extend the functionality of the operation.

Operation consists of 2 operation types at the current stage:

1. ChangeLockOP

   ChangeLockOP is used to change relevant Passport's lock. ChangeLockOP has an OP ID of 0 and a lock which is the new lock for the current Passport.

2. ReservedNonceOP

   ReservedNonceOP is used to reserve some nonces for future transactions, so that these reserved nonces can be used to execute some offchain transactions (such as EQC lightning network transactions), and these reserved nonces can be used to execute transactions update the relevant global state on the EQC network when needed.

   ReservedNonceOP has an OP ID of 1 and a reserved nonce quantity(Values range from 1 to 256), which is the number of nonces reserved.

   When the ReservedNonceOP is executed, the following operations will be performed:

1. The nonce of the current Passport will increase the number of reserved nonces.
2. The reserved nonce flag will be marked in the corresponding Passport's status state object (if necessary), and the value of the total number of reserved nonce state object will be increased according to the total number of reserved nonces, and the corresponding reserved nonces' value will be added to the reserved nonce ZeroOneDynamicMerklePatriciaTrie.

Transaction consists of four body types at the current stage:

1. IssuePassportBody

   IssuePassportBody is used to issue passports. A maximum of 129 IssuePassportBodies can be included in the IssuePassportBody array.

2. OperationBody

   OperationBody is used to execute operation, for example ChangeLockOP. A maximum of 65 OperationBodies can be included in the OperationBody array.

3. TransferBody

   TransferBody is used to transfer EQC. A maximum of 257 TransferBodies can be included in the TransferBody array.

4. SmartContractBody

SmartContractBody is used to execute smart contract function. A maximum of 5 SmartContractBodies can be included in the SmartContractBody array.

2.8.3 Transaction storage structure

The transaction storage structure consists of two state objects: L(length) state object(the transaction length, its data type is EQCBits，its count index does not start from 0 but from 69 (because the size of the smallest transaction is 69 bytes)) and V(value) state object(the transaction body).

2.9 Transaction use cases

2.9.1 Issue Passport

1. Adam issues passport and transfers 101 EQC for Eve (Lock: 0xbb...bb).

Before Adam sends the transaction：

Adam's Passport
Status: 0
ID: 1
Balance: 2100000000000000
Nonce: 0

LockNonce: 1
Lock: 0xaa...aa
PublicKey: null

Transaction sent by Adam to issue Passport:

Transaction	
Status[1]: 00000001[2]	
Passport ID: 1	
Nonce: 1	
IssuePassportBody	Status[3]: 0000000[4]0[5]
	Lock: bb...bb[6]
	Value: 10100000000
Signature: xx...xx	

The size of the transaction is 105 bytes.

After Adam sends the transaction:

---

[1] The type of the Status state object is EQCBits.

[2] Indicates whether transaction includes IssuePassportBody, 0: excludes, 1: includes.

[3] The type of the Status state object is EQCBits.

[4] This state object includes a series of consecutive bits. When IssuePassportBody includes only one sub-IssuePassportBody uses it to record the lock type part of the current sub-IssuePassportBody's lock, and when IssuePassportBody includes multiple sub-IssuePassportBodys uses it to record the number of sub-IssuePassportBodys. The current record value is the lock type part 0000000(0) of the sub-IssuePassportBody's lock.

[5] Indicates whether IssuePassportBody includes multiple sub-IssuePassportBodys, 0: one, 1: multiple.

[6] When IssuePassportBody includes only one sub-IssuePassportBody uses it to record the hash part of the lock, and when IssuePassportBody includes multiple sub-IssuePassportBodys uses it to record the full lock. The current record value is the public key hash part bb...bb of sub-IssuePassportBody's T0 lock.

Adam's Passport
Status: 0
ID: 1
Balance: 2099989800090000
Nonce: 1
LockNonce: 1
Lock: 0xaa...aa
PublicKey: xx...xx

Eve's Passport
Status: 0
ID: 2
Balance: 10100000000
Nonce: 0
LockNonce: 1
Lock: 0xbb...bb
PublicKey: null

2. Adam issues passports and transfers 101 EQC for Moses (Lock: 0xcc...cc) and Noah (Lock: 0xdd...dd) and charge the service fee of 1 EQC per person. Therefore, after deducting the service fee, the transfer amount is 100 EQC.

Before Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099989800090000
Nonce: 1
LockNonce: 1
Lock: 0xaa...aa
PublicKey: xx...xx

Transaction sent by Adam to issue Passports:

Transaction	
Status: 00000001	
Passport ID: 1	
Nonce: 2	
IssuePassportBody	Status: 0000000[7]1[8]
	Lock: 0xcc...cc
	Value: 10000000000
	Lock: 0xdd...dd
	Value: 10000000000
Signature: xx...xx	

The size of the transaction is 144 bytes.

[7] Indicates IssuePassportBody includes multiple sub-IssuePassportBodys.

[8] Record the current number of sub-IssuePassportBodys is 0000000(2).

After Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099969800080000
Nonce: 2
LockNonce: 1
Lock: 0xaa...aa
PublicKey: xx...xx

Moses' Passport
Status: 0
ID: 3
Balance: 10000000000
Nonce: 0
LockNonce: 1
Lock: 0xcc...cc
PublicKey: null

Noah's Passport
Status: 0

ID: 4
Balance: 10000000000
Nonce: 0
LockNonce: 1
Lock: 0xdd...dd
PublicKey: null

2.9.2 Transfer

1. Adam transfers 101 EQC to Moses.

Before Adam sends the transaction：

Adam's Passport
Status: 0
ID: 1
Balance: 2099969800080000
Nonce: 2
LockNonce: 1
Lock: 0xaa...aa
PublicKey: xx...xx

Transaction sent by Adam to transfer：

Transaction
Status: 0000[9]1[10]000

[9] When transaction includes TransferBody indicates whether TransferBody includes multiple

Passport ID: 0	
Nonce: 3	
TransferBody[11]	Passport ID: 2
	Value[12]: 10100000000
Signature: xx...xx	

The size of the transaction is 72 bytes.

After Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099959700070000
Nonce: 3
LockNonce: 1
Lock: 0xaa...aa

---

sub-TransferBodys, 0: one, 1: multiple, otherwise indicates whether transaction includes SmartContractBody, 0: excludes, 1: includes.

[10] Indicates whether transaction includes TransferBody, 0: excludes, 1: includes.

[11] ~~EQcoin uses EQCHelix to store the transfer value and relevant Passport ID's bytes' length in TransferBody.~~ On the underlying storage, Value is stored first, followed by Passport ID.

[12] The lowest 5 bits of the binary digits of Value are reserved as identifier bits to store the number of bytes occupied by Value and Passport ID respectively, among which the upper 3 bits are used to store the number of bytes occupied by Value and the lower 2 bits are used to store the number of bytes occupied by Passport ID. In the TransferBody, Value is stored first and then Passport ID is stored. Therefore, if the value of the transfer amount entered in TransferBody is not divisible by 32, some adjustments need to be made to make it divisible by 32 so that the lowest 5 bits of its binary digits are reserved as identifier bits. It is recommended to adjust it in the following ways: 1. Use the result of value-(value%32) as the transfer amount. 2. Use the result of value+(32-(value%32)) as the transfer amount.

PublicKey: xx...xx

Moses' Passport
Status: 0
ID: 3
Balance: 20100000000
Nonce: 0
LockNonce: 1
Lock: 0xcc...cc
PublicKey: null

2. Adam transfers 101 EQC to Moses and Noah.

Before Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099959700070000
Nonce: 3
LockNonce: 1
Lock: 0xaa...aa
PublicKey: xx...xx

Transaction sent by Adam to transfer:

Transaction

Status: 0001[13]1000	
Passport ID: 0	
Nonce: 4	
TransferBody	Array length: 00000000[14]
	Passport ID: 3
	Value: 10100000000
	Passport ID: 4
	Value: 10100000000
Signature: xx...xx	

The size of the transaction is 78 bytes.

After Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099939500060000
Nonce: 4
LockNonce: 1
Lock: 0xaa...aa
PublicKey: xx...xx

[13] Indicates TransferBody includes multiple sub-TransferBodys.

[14] Record the current number of sub-TransferBodys is 00000000(2).

Moses' Passport
Status: 0
ID: 3
Balance: 30200000000
Nonce: 0
LockNonce: 1
Lock: 0xcc...cc
PublicKey: null

Noah's Passport
Status: 0
ID: 4
Balance: 20100000000
Nonce: 0
LockNonce: 1
Lock: 0xdd...dd
PublicKey: null

2.9.3 Change lock

1. Adam changes his Passport's lock to 0bb...bb.

Before Adam sends the transaction：

Adam's Passport
Status: 0
ID: 1
Balance: 2099939500060000
Nonce: 4
LockNonce: 1
Lock: 0xaa...aa
PublicKey: xx...xx

Transaction sent by Adam to change lock:

Transaction	
Status: 000000[15]1[15]0	
Passport ID: 0	
Nonce: 5	
OPBody	Status[16]: 0000000[17]0[18]
	Lock: 0xbb...bb[19]
Signature: xx...xx	

---

[15] Indicates whether transaction includes OPBody, 0: excludes, 1: includes.

[16] The type of the Status state object is EQCBits.

[17] This state object includes a series of consecutive bits. When OPBody includes only one sub-OPBody uses it to record the OP ID part of the sub-OPBody, and when OPBody includes multiple sub-OPBodys uses it to record the number of OPBodys. The current record value is the OP ID part 0000000(0) of the ChangeLockOP.

[18] Indicates whether OPBody includes multiple sub-OPBodys, 0: one, 1: multiple.

[19] When OPBody includes only one sub-OPBody uses it to record the OP body part of the OP, and when OPBody includes multiple sub-OPBodys uses it to record the full OP. The current record value is the lock part 0bb...bb of ChangeLockOP.

The size of the transaction is 101 bytes.

After Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099939500050000
Nonce: 5
LockNonce: 2
Lock: 0xbb...bb
PublicKey: null

2.9.4 Execute smart contract

1. Adam executes the Buy function (ID: 4) of the EQswap SmartContract (ID: 1002) and uses 0.00000051 EQC to buy 201 Bethard tokens from Bethard.

Before Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099939500050000
Nonce: 5
LockNonce: 2

Lock: 0xbb...bb
PublicKey: null

Transaction sent by Adam to execute smart contract:

Transaction	
Status: 000[20]1[21]0000	
Passport ID: 0	
Nonce: 6	
SmartContractBody	Status: 00100[22]01[23]0[24]
	SmartContract ID: 1002
	Function ID: 4[25]
	Value[26]: 51
Signature: xx...xx	

[20] When transaction includes SmartContractBody indicates whether SmartContractBody includes multiple sub-SmartContractBodys, 0: one, 1: multiple, otherwise reserved for indicates other states of the transaction.

[21] Indicates whether transaction includes SmartContractBody, 0: excludes, 1: includes.

[22] When the current smart contract function is called once uses it to record the function ID, and when the current smart contract function is called multiple times uses it to record the number of calls. The current record value is the current smart contract function ID 00100(4) which is Buy function.

[23] Record the bytes of the Passport ID in the smart contract ID. The current record value is the size of the byte stream corresponding to Passport ID 1002, which is 01(2) bytes.

[24] Indicates whether current smart contract function includes multiple calls, 0: one, 1: multiple.

[25] The value is saved in the Status state object identified by note 21.

[26] The lowest 3 bits of the binary digits of Value are reserved as identifier bits to store the number of bytes occupied by Value. Therefore, if the value of the transfer amount entered in SmartContractBody is not divisible by 8, some adjustments need to be made to make it divisible by 8 so that the lowest 3 bits of its binary digits are reserved as identifier bits. It is recommended to adjust it in the following ways: 1. Use the result of value-(value%8) as the transfer amount. 2. Use the result of value+(8-(value%8)) as the transfer amount.

The size of the transaction is 72 bytes.

After Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099939500039949
Bethard token: 20100000000
Nonce: 6
LockNonce: 2
Lock: 0xbb...bb
PublicKey: xx...xx

2. Adam executes the Buy function (ID: 4) of the EQswap SmartContract (ID: 1002) and uses 0.00000051 EQC to buy 201 Bethard tokens from Bethard, then executes the betting function (ID: 3) of the Bethard horse racing SmartContract (ID: 1003) and uses 201 EQC to bet that No. 9 of the Royal Ascot's Her Majesty's Plate will win.

Before Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099939500039949

Bethard token: 20100000000
Nonce: 6
LockNonce: 2
Lock: 0xbb...bb
PublicKey: xx...xx

Transaction sent by Adam to execute smart contract:

Transaction	
Status: 001[27]10000	
Passport ID: 1	
Nonce: 7	
SmartContractBody	Status: 000000[28]00[29]
	Status: 00100010
	Smart contract ID: 1002
	Function ID: 4
	Value: 51
	Status: 00011010
	Smart contract ID: 1003
	Function ID: 3
	Value: 20100000000
	Winner: 9

[27] Indicates SmartContractBody includes multiple sub-SmartContractBodys.

[28] Reserved status flag bits.

[29] Record the number of sub-SmartContractBodys. The current record value is 00(2).

Signature: xx...xx

The size of the transaction is 83 bytes.

After Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099919400029898
Bethard token: 40200000000
Nonce: 7
LockNonce: 2
Lock: 0xbb...bb
PublicKey: xx...xx

3. Adam executes the Transfer function (ID: 1) of the Bethard token contract (ID: 1003 and transfer 100 Bethard tokens to Eve, Moses and Noah.

Before Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099919400029898
Bethard token: 40200000000
Nonce: 7

LockNonce: 2
Lock: 0xbb...bb
PublicKey: xx...xx

Transaction sent by Adam to execute smart contract:

Transaction	
Status: 00100000	
Passport ID: 1	
Nonce: 8	
SmartContractBody	Status: 00001[30]011[31]
	Smart contract ID: 1003
	Function ID: 1
	Passport ID: 2
	Value: 100
	Passport ID: 3
	Value: 100
	Passport ID: 4
	Value: 100
Signature: xx...xx	

The size of the transaction is 77 bytes.

After Adam sends the transaction:

[30] The current record value is the number of current smart contract function calls: 00001(3).

[31] Indicates current smart contract function includes multiple calls.

Adam's Passport
Status: 0
ID: 1
Balance: 2099919400019898
Bethard token：10200000000
Nonce: 8
LockNonce: 2
Lock: 0xbb...bb
PublicKey: xx...xx

Eve's Passport
Status: 0
ID: 2
Balance: 10100000000
Bethard token： 10000000000
Nonce: 0
LockNonce: 1
Lock: 0xbb...bb
PublicKey: null

Moses' Passport
Status: 0

ID: 3
Balance: 30200000000
Bethard token：10000000000
Nonce: 0
LockNonce: 1
Lock: 0xcc...cc
PublicKey: null

Noah's Passport
Status: 0
ID: 4
Balance: 20100000000
Bethard token：10000000000
Nonce: 0
LockNonce: 1
Lock: 0xdd...dd
PublicKey: null

2.9.5 Complex transaction

Adam issues passport and transfers 51 EQC for Amon（Lock: 0xee...ee, transfers 101 EQC to Moses, executes the Buy

function (ID: 4) of the EQswap smart contract (ID: 1002) and uses 0.00000051 EQC to buy 201 Bethard tokens from Bethard and changes his Passport's lock to 0ff...ff and set the power price to 11 to execute transactions at a faster accounting rate.

Before Adam sends the transaction:

Adam's Passport
Status: 0
ID: 1
Balance: 2099919400019898
Bethard token: 10200000000
Nonce: 8
LockNonce: 2
Lock: 0xbb...bb
PublicKey: xx...xx

Complex Transaction sent by Adam:

Transaction	
Status: 001011[32]11	
Passport ID: 1	
Nonce: 9	
IssuePassportBody	Status: 00000000
	Lock: ee...ee

[32] Indicates whether transaction specifies a higher power price, 0: default, 1: higher.

	Value: 5100000000
	Lock: 0xff...ff
TransferBody	Passport ID: 2
	Value: 10100000000
SmartContractBody	Status: 00100010
	Smart contract ID: 1002
	Function ID： 4
	Value: 51
OPBody	Status: 00000000
	Lock: 0xff...ff
PowerPrice	Value: 11
Signature: xx...xx	

The size of the transaction is 151 bytes.

After Adam sends the transaction：

Adam's Passport
Status: 0
ID: 1
Balance: 2099904200008847
Bethard token：30300000000
Nonce: 9
LockNonce: 2
Lock: 0xff...ff

PublicKey: null

Moses' Passport
Status: 0
ID: 3
Balance: 40300000000
Bethard token: 10000000000
Nonce: 0
LockNonce: 1
Lock: 0xcc...cc
PublicKey: null

Amon's Passport
Status: 0
ID: 5
Balance: 5100000000
Nonce: 0
LockNonce: 1
Lock: 0xee...ee
PublicKey: null

2.10 About MerklePatriciaTrie

2.10.1 About ZeroOneMerklePatriciaTrie and ZeroOneDynamicMerklePatriciaTrie

ZeroOneMerklePatriciaTrie is used to store the Global State of the IDKey storage object(for example Passport 、Transaction 、active SmartContract relevant state objects) in each block of EQcoin. The IDKey state object has a unique ID, which is a natural number encoded consecutively from one. In the ZeroOneMerklePatriciaTrie, it is bit by bit addressed from high bit to low bit according to the binary value of the ID of the relevant IDKey state object. ZeroOneMerklePatriciaTrie includes two types of keys, ZeroKey(0) and OneKey(1) , they are consist of only one bit(0 or 1).

Note： Due to the ZeroKey and OneKey contain only one character, there is no need to store it, and it can be obtained directly according to the corresponding status bit of its parent node.

ZeroOneDynamicMerklePatriciaTrie is used to store the Global State of the IDKey state objects(for example active and inactive Smart Contract relevant state objects) in each block of EQcoin. In the ZeroOneDynamicMerklePatriciaTrie, it is bit by

bit addressed from high bit to low bit according to the binary value of the Hash of the relevant state object. ZeroOneDynamicMerklePatriciaTrie includes two types of keys, ZeroDynamicKey(0xxx) and OneDynamicKey(1xxx) , they are consecutive binary sequences consisting of one or more bits starting with 0 and 1.

Note : When the ZeroDynamicKey and OneDynamicKey contain only one character, there is no need to store it, and it can be obtained directly according to the corresponding status bit of its parent node. When the ZeroDynamicKey and OneDynamicKey contain multiple characters, there is no need to store its first character because it can be obtained directly according to the corresponding status bit of its parent node.

2.10.1.1 About BinaryNode

BinaryNode is the key node that constitutes the ZeroOneMerklePatriciaTrie or ZeroOneDynamicMerklePatriciaTrie dictionary tree. BinaryNode has a status, a key, a ZeroNode, a OneNode and a value. ZeroOneMerklePatriciaTrie includes only two BinaryNodes, ZeroNode and OneNode. BinaryNode's underlying storage implementation includes a HashKey(Hash

of BinaryNode's binary raw data, used to support state object verification based on light client protocol) and a StorageKey(UpdateNonce(A natural number starting from 0 and increments by 1 with each modification of the BinaryNode) of BinaryNode, used to access state objects from StateDB).

2.10.1.2 About BinaryNode Status

BinaryNode Status is used to identify the composition of the state objects included in the BinaryNode. The type of the BinaryNode Status state object is EQCBits.

The default order of state objects that BinaryNode includes is: ZeroNode, OneNode, and Value.

BinaryNode Status consists of two status types at the current stage(in the underlying storage, they are compositely stored together):

1. HashStatus, the default universal identifier bit of the BinaryNode which participate in the calculation of the BinaryNode Hash.

00000[33]0[34]0[35]

2. StorageStatus, includes HashStatus identifier bit and storage relvant identifier bit of the BinaryNode which does not participate in the calculation of the BinaryNode Hash but it is

---

[33] Indicates whether node includes OneNode, 0: excludes, 1: includes.

[34] Indicates whether node includes ZeroNode, 0: excludes, 1: includes.

[35] Indicates whether node includes Value, 0: excludes, 1: includes.

used to get BinaryNode from StateDB.

2.1 When BinaryNode includes two state objects(ZeroNode&Value or OneNode&Value).

0000[36]0[37]000[38]

Note: The state object has the smallest StorageKey hereinafter referred to as “A” (when there are multiple equal value minimums, the one with the lowest default order is taken), another state object hereinafter referred to as “B”. If A's StorageKey equals B's StorageKey, the underlying data is A's StorageKey . If A's StorageKey is not equal to B's StorageKey, the underlying data is A's StorageKey and (B's StorageKey - A's StorageKey)(this saves more storage space than directly stores the two state objects'storageKey). For example, if B's StorageKey is 100,001 and the A's StorageKey is 100000, the underlying stored data is 100,000 and 1(this saves a lot of storage space than direct storage 100,000 and 100,001). When need to restore the B's StorageKey, can obtain its value through 100000+1.

2.2 When BinaryNode includes three state objects(ZeroNode, OneNode and Value).

---

[36] Identifies whether the two state objects'storageKey are equal, 0: no, 1: yes.

[37] Identifies which node state object'storageKey is the smallest in the default order, 0: left, 1: right.

[38] These 3 identifier bits are the same as in HashStatus.

0[39]0[40]0[41]00[42]000[43]

Note: The state object has the smallest StorageKey hereinafter referred to as “A” (when there are multiple equal minimum values, the one with the lowest default order is taken), the state object after A in the default order hereinafter referred to as “B” (If A is Value, the order is calculated from the beginning, so B is ZeroNode) , the state object after B in the default order hereinafter referred to as “C” (If B is Value, the order is calculated from the beginning, so C is ZeroNode) . The one with the smaller StorageKey in B and C hereinafter referred to as “D” (when the StorageKey of B and C are equal, the one with the lowest default order is taken)and the The one with the larger StorageKey in B and C hereinafter referred to as “E” (when the StorageKey of B and C are equal, the one with the highest default order is taken).

2.10.1.3 About ZeroNode

As shown in the figure below, the key of ZeroNode is 0.

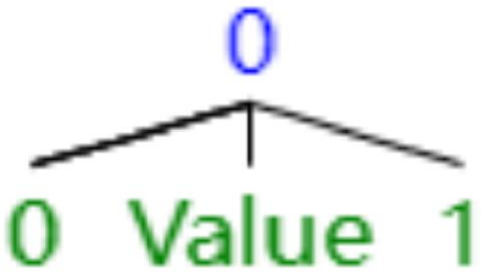
[39] Identifies whether E's StorageKey is equal to D's StorageKey, 0: no, 1: yes.

[40] Identifies whether D's StorageKey is equal to A's StorageKey, 0: no, 1: yes.

[41] Indicates B's and C's StorageKey who is bigger, 0: B<=C, 1: B>C.

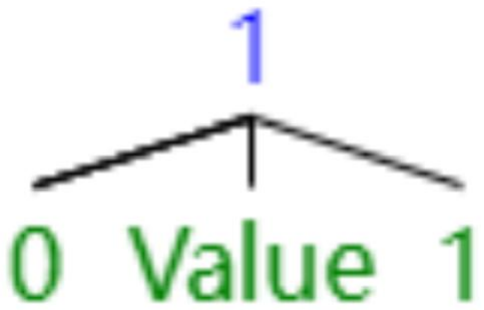
[42] Indicates which state object has the smallest UpdateNonce, 0: ZeroNode, 1: OneNode, 2: Value.

[43] These 3 identifier bits are the same as in HashStatus.



2.10.1.4 About OneNode

As shown in the figure below, the key of OneNode is 1.



2.10.2 About HexMerklePatriciaTrie

HexMerkleDynamicPatriciaTrie is used to store the Global State of the HashKey state object(for example Smart Contract relevant state objects) in each block of EQcoin. ~~In the HexMerklePatriciaTrie, it is bit by bit addressed from high bit to low bit according to the binary value of the Hash of the relevant state object~~. HexMerklePatriciaTrie includes 16 keys,

DynamicKey0(0xxx), DynamicKey1(1xxx), DynamicKey2(2xxx), DynamicKey3(3xxx), DynamicKey4(4xxx), DynamicKey5(5xxx), DynamicKey6(6xxx), DynamicKey7(7xxx), DynamicKey8(8xxx), DynamicKey9(9xxx), DynamicKeyA(Axxx), DynamicKeyB(Bxxx), DynamicKeyC(Cxxx), DynamicKeyD(Dxxx), DynamicKeyE(Exxx), DynamicKeyF(Fxxx), they are continuous hexadecimal string keywords starting from 0 to F respectively.

2.10.2.1 About BranchNode

BranchNode is the key node that constitutes the HexMerkleDynamicPatriciaTrie dictionary tree. BranchNode has a status, a key, a BranchNode0, a BranchNode1, a BranchNode2, a BranchNode3, a BranchNode4, a BranchNode5, a BranchNode6, a BranchNode7, a BranchNode8, a BranchNode9, a BranchNodeA, a BranchNodeB, a BranchNodeC, a BranchNodeD, a BranchNodeE, a BranchNodeF, and a Leaf. HexMerkleDynamicPatriciaTrie includes 16 BranchNodes, BranchNode0, BranchNode1, BranchNode2, BranchNode3, BranchNode4, BranchNode5, BranchNode6, BranchNode7, BranchNode8, BranchNode9, BranchNodeA, BranchNodeB, BranchNodeC, BranchNodeD, BranchNodeE and BranchNodeF.

BranchNode's underlying storage implementation includes a HashKey(Hash of BranchNode's binary raw data, used to support state object verification based on light client protocol) and a StorageKey(UpdateNonce(A natural number starting from 0 and increments by 1 with each modification of the BranchNode) of BranchNode, used to access state objects from StateDB).

2.10.1.2 About BranchNode Status

BranchNode Status is used to identify the composition of the state objects included in the BranchNode. The type of the BranchNode Status state object is EQCBits.

The default order of state objects that BranchNode includes is: BranchNode0，BranchNode1，BranchNode2，BranchNode3，BranchNode4，BranchNode5，BranchNode6，BranchNode7，BranchNode8，BranchNode9，BranchNodeA，BranchNodeB，BranchNodeC，BranchNodeD，BranchNodeE, BranchNodeF, and Value.

BranchNode Status consists of two status types at the current stage(in the underlying storage, they are compositely stored together):

1. HashStatus, the default universal identifier bit of the BranchNode which participate in the calculation of the BranchNode Hash.

0[44]**0**[45]**0**[46]**0**[47]**0**[48]**0**[49]**0**[50]**0**[51]**0**[52]**0**[53]**0**[54]**0**[55]**0**[56]**0**[57]**0**[58]**0**[59]**0**[60]**0**[61]

2. StorageStatus, includes HashStatus identifier bit and storage relvant identifier bit of the BranchNode which does not participate in the calculation of the BranchNode Hash but it is used to get BranchNode from StateDB.

0[62]**0**[63]**0xxx0**[64]**0000xxx0000**[65]0000000000000000

---

[44] Indicates whether BranchNode includes BranchNodeF, 0: excludes, 1: includes.

[45] Indicates whether BranchNode includes BranchNodeE, 0: excludes, 1: includes.

[46] Indicates whether BranchNode includes BranchNodeD, 0: excludes, 1: includes.

[47] Indicates whether BranchNode includes BranchNodeC, 0: excludes, 1: includes.

[48] Indicates whether BranchNode includes BranchNodeB, 0: excludes, 1: includes.

[49] Indicates whether BranchNode includes BranchNodeA, 0: excludes, 1: includes.

[50] Indicates whether BranchNode includes BranchNode9, 0: excludes, 1: includes.

[51] Indicates whether BranchNode includes BranchNode8, 0: excludes, 1: includes.

[52] Indicates whether BranchNode includes BranchNode7, 0: excludes, 1: includes.

[53] Indicates whether BranchNode includes BranchNode6, 0: excludes, 1: includes.

[54] Indicates whether BranchNode includes BranchNode5, 0: excludes, 1: includes.

[55] Indicates whether BranchNode includes BranchNode4, 0: excludes, 1: includes.

[56] Indicates whether BranchNode includes BranchNode3, 0: excludes, 1: includes.

[57] Indicates whether BranchNode includes BranchNode2, 0: excludes, 1: includes.

[58] Indicates whether BranchNode includes BranchNode1, 0: excludes, 1: includes.

[59] Indicates whether BranchNode includes BranchNode0, 0: excludes, 1: includes.

[60] Indicates whether BranchNode's key is one character or multiple characters, 0: one, 1: multiple. When the key contains only one character, there is no need to store it, and it can be obtained directly according to the corresponding status bit of its parent HashNode. When the key contains multiple characters, there is no need to store its first character because it can be obtained directly according to the corresponding status bit of its parent HashNode.

[61] Indicates whether BranchNode is a branch node or a leaf node, 0: branch, 1: leaf.

[62] If Value exists indicates the absolute value of the difference between Value's StorageKey and the StorageKey of the largest or smallest BranchNode which is smaller, 0: the smallest, 1: the largest.

[63] Identifies whether the value's StorageKey is larger or smaller, 0: smaller, 1: larger.

[64] The 4-bit identifier identifies the sequence number of each BranchNode included in the current node sorted from small to large(when the StorageKeys of two adjacent BranchNodes are equal, the one with the lowest default order is taken).

[65] The one-bit flag identifies whether each BranchNode(from the second BranchNode sorted in ascending order) is equal to the adjacent BranchNode that is smaller than it.

Note： If Value exists, the smaller absolute value of the difference between its StorageKey and the StorageKey of the largest or smallest BranchNode will be stored.

2.10.2.3 About LeafNode

LeafNode is the leaf node that constitutes the HexMerkleDynamicPatriciaTrie dictionary tree. LeafNode is used to store state objects. LeafNode has a status, a **StateObjectMate or StateObjectMateArray**[66] and its relevant governance state objects. LeafNode related governance state objects can be extended as needed through its status state object. A **HashKey collisionless identifier**[67] bit is included in the current status to identify whether the current LeafNode includes multiple state objects with the same HashKey.

2.10.2.3.1 About StateObjectMate

StateObjectMate is used to store state object and its relevant state objects. StateObjectMate has a status, a specific

---

[66] When the current LeafNode contains only one state object, only one StateObjectMate object is included. When the current LeafNode includes more than two state objects, the current LeafNode includes a StateObjectMateArray (its array subscript 0 represents 2 array elements, array subscript 1 represents 3 array elements, and so on).

[67] HashKey collisionless identifier identifies whether the current LeafNode includes state objects that have collisions, 0: collisionless, 1: collision.

state object and its relevant state objects. StateObjectMate related governance state objects can be extended as needed through its status state object.

2.10.2.3.2 About StateObjectMateArray

StateObjectMateArray is used to simultaneously store multiple StateObjectMates, which contain state objects with the same HashKey. These state objects can be identified and distinguished by their UUID(Universally Unique Identifier)s, which are generated by specific algorithms based on the type, raw data, and associated unique identifiers of specific state objects.

2.10.2.3.3 The HashKey collisionless design of HexMerklePatriciaTrie

Before accessing a specific state object in the HexMerklePatriciaTrie, the global unique access lock bound to its HashKey must be obtained first, and the related state object can be read, written and deleted only after the access right of the access lock is obtained. After the access operation of the relevant state object is completed, its access lock needs to be released.

Read operation:

When performing a read operation, if the current HashKey does not exist, null is returned directly.

When performing a read operation, if the current LeafNode does not have a HashKey collision, it will directly return it including StateObjectMate, and then obtain the corresponding state object from the StateObjectMate according to the UUID provided by the current read operation.

When performing a read operation, if the current LeafNode has a HashKey collision, the StateObjectMateArray included in it will be returned, and then obtains the corresponding state object from the StateObjectMateArray according to the UUID provided by the current read operation.

Write operation:

When performing a write operation, if the current HashKey does not exist, the current state object is directly stored in the corresponding LeafNode.

When performing a write operation, if the current HashKey exists, and if the current LeafNode does not have a HashKey collision, then compare whether the UUID of the state object contained in the StateObjectMate currently contained in it is

consistent with the UUID of the state object to be stored? If the UUIDs are consistent, the corresponding storage is directly overwritten. Otherwise, if the UUIDs are inconsistent, this is a HashKey collision. In this case, the HashKey collisionless identifier of the current LeafNode will be marked as 1 and the StateObjectMateArray will be used to simultaneously store the two state objects.

When performing a write operation, if the current HashKey exists, and if the current LeafNode has a HashKey collision, then compare the UUID of the state object stored in the current StateObjectMateArray one by one with the UUID of the state object to be stored. If the UUID of a state object stored in the current StateObjectMateArray is consistent with the UUID of the state object to be stored, the corresponding state object is directly overwritten and stored. If the UUIDs of all the state objects stored in the current StateObjectMateArray are inconsistent with the UUID of the state object to be stored, add a new StateObjectMate array element containing the state object currently to be stored in the StateObjectMateArray to store it.

Delete operation：

When performing a delete operation, if the current HashKey does not exist, do nothing.

When performing a delete operation, if the current HashKey exists, and if the current LeafNode does not have a HashKey collision, then compare whether the UUID of the state object contained in the StateObjectMate contained in it is consistent with the UUID provided by the current delete operation. If they are consistent, delete the current LeafNode, otherwise do nothing.

When performing a delete operation, if the current HashKey exists, and if the current LeafNode has a HashKey collision, then compare the UUID of the state object stored in the current StateObjectMateArray one by one with the UUID of the state object to be deleted. If the UUID of a state object stored in the current StateObjectMateArray is consistent with the UUID of the state object to be deleted, the corresponding StateObjectMate is directly deleted(If the current StateObjectMateArray contains only one StateObjectMate after the delete operation, then mark the HashKey collisionless identifier of the current LeafNode as 0, and delete the current StateObjectMateArray then store the StateObjectMate it contains directly in the LeafNode), otherwise do nothing.

2.10.2.3.4 Pre-research on the Design of Key Storage for Merkle Patricia Trie

Questions and Objectives

problem

StateDB is a KV database.

There are two options for selecting keys for each node in MPT: hash based keys (which have a huge space cost and may collide) and nonce based keys (which allocate space as needed and do not collide).

MPT must provide hash locking function, so each node will have 2 keys. One non based storage key is used to store nodes in StateDB, and the other is a hash based hash key used to lock related state objects.

target

Design an MPT storage key scheme that minimizes state data and eliminates key collisions.

background knowledge

Related research and comparative evaluation analysis optimization discussion

StorageObject has a unique StorageKey in the current cache used as its storage key, which not only avoids the possibility of collision and value overwriting when using State

Object's hash as the storage key, but also saves storage space.

There is no need to use a storage key for transactions, as transactions are unique and will not change. But the passport is different. Each height will have some changes, so using storage keys has advantages, which can save storage space for unchanged objects. Because storing keys is much smaller than hashing.

Continuing optimization design of Key in MPT tree: 1 Change the UpdateHeight of a specific node to the Updatenonce of the node (or rename it as revision?) to further save space. After all, with the extension of blockchain, many nodes may not necessarily have updates in every block. 2. Compress the Storage Key of the storage node? The problem is that while compressing, it is also necessary to ensure its global uniqueness.

The current StorageKey based storage structure for MPT nodes and LeafNodes has problems with the key value. It not only does not save space but also increases space (adding an additional StorageKey storage space because each node's HashKey still needs to be stored, so MinorityReport needs to be added to solve the hash collision problem. If a collision occurs, the Status field needs to be added for each node. If a

collision occurs, the MR tag bit needs to be marked, and an MR object needs to be added to the tail of the associated object for this node (EQCBits form the first non collision value from zero). However, this may make parallel computing difficult to perform during SaveMPT.

Note: Use the Eof at the end of the Node to determine if there is a collision. If Eof is encountered at the tail when parsing the current node, it proves that the elements included in the current node do not have collisions. Otherwise, it proves that the elements included in the current node have collisions. At this point, the MinorityReport object is included at the tail of the node. Its composition is as follows: Status field (a byte, bit 0: whether LN is colliding, bit 1: whether RN is colliding, bit 2: whether Value is colliding), nonce field (each nonce records the nonce value that is stacked in parallel on the hash tail of the current LN, RN, and Value to jointly form a specific Node's Key that does not collide for the first time). This solution decouples and combines the UniqueKey of HashKey for MPT collisions and the OriginalKey that makes up MPT, allowing for convenient parallel and concurrent storage.

Note: MPT will be implemented through a thread safe VirginPut function during put. It is best for KVDB to also

support functions similar to VirginPut to achieve the following functions: put will check whether there is a corresponding key in the underlying index. If it exists, terminate the put operation and return false. If it does not exist, put will be successful and return true. Otherwise, for new objects, we can only use the get operation on their key to see if they exist. If they do not exist, we can directly put them in. If they do exist, it indicates that a collision has been sent. At this point, after the original key, we first stack the increasing nonces one by one and then get them again until the composite key does not exist. After that, we can put the composite key and store the nonces.

Note: This solution solves the problem of object overwrite caused by storing state objects with the same key, but it does not solve the problem of object replacement. That can (should) be solved by waiting for enough confirmation to obtain the corresponding state object data from the trusted full node based on the light client protocol.

Note: The KV DB of EQC adopts the following safety model: 1 Sandbox 2 Unique key

Compare and analyze the TCO of StorageKey and HashKey under different update states of child elements in the current node.

By using the relative comparison method, the order can be determined by using 2 digits. Without recording ln, first record and compare the sizes of rn and ln, with the status markers indicating which one is larger or smaller, 0 being smaller and 1 being larger, and the high-order markers indicating whether the two are the same. Then, based on the comparison results and leaf, if different, take the larger of the two?

But it is possible to determine the order by only recording rn and leaf with 3 bits each.

StorageKey design. Use block height as the storage key for nodes with changes. And mark in the Status whether the current specific node has an update. If there is u, its value is the height of the current block. If there is no change, its value must be lower than the current height, so store its difference. Every time MPT is opened, it will restore the storage key of a specific node based on its previous state, so that a new difference can be calculated based on the previous storage key and its current state. This is not feasible because as the height increases, it will lead to a change in the current base

height, resulting in the loss of the original base heights of old nodes that have not changed.

Conclusion: It is not feasible to use the height of the current block as the baseKey, as the update height of the node that has not been updated can only be obtained at its updated height, and all subsequent blocks can only obtain the current block height directly based on the flag in the Status. Therefore, each node must store a baseKey to solve this problem. Therefore, it is still necessary to use a Nonce based storage key mechanism.

The value of the current node must definitely exist, but its LN and RN may not necessarily both exist. Except for the outermost layer of each branch, all other nodes LN, RN, and value exist. However, the normative requirement based on data independence still requires the storage of flag bits for Value in Status (a total of 6 status values require 3 bits).

LN, RN, and value have the following combination of relationships:

LN+Value.

RN+Value.

LN+RN+value.

The nonce of LN, RN, and value has the following relationship combination (a total of three state values need to occupy 2 bits):

1. The three are not the same (only the minimum and maximum values need to be recorded (according to the exclusion method, only the middle value can be left), a total of 4 bits are required).

Use case: The current height only updates one child node of a specific node, and the update heights of the other two child nodes are not the same.

2. All three are the same.

Use case: The current height simultaneously updates all three child nodes of a specific node.

Two of them are the same in 3.3 (only the minimum value needs to be recorded (according to the exclusion method, the remaining can only be the maximum value), totaling 2 bits.

Use case: The current height simultaneously updates two child nodes of a specific node (or after updating one child node, its nonce is the same as one of the other two child nodes), and the updated height of the remaining child node is different from the other two child nodes.

3.1 The following possibilities exist

LN=Value

RN=Value

LN=RN

In Status, existence is first recorded, followed by identity relationships, and then whose nonce is the smallest and whose different nodes are larger and smaller are recorded.

Status bit design

Number 0 bit: Does the Value exist.

No.1 bit: Does LN exist.

Bit 2: Whether RN exists.

No.3 bit: Is the Value the smallest.

No.1 bit: Is LN the smallest.

Bit 2: Whether RN exists.

Analysis of Status Bit Occupation:

3 (State of existence of LN, RN, and Value)+2 (State of combination of LN, RN, and value size relationships (all three are the same, all three are different, where two are the same))+4 (State of LN, RN, and value size sorting: all three are different)

3+2+0 (all three are the same)

3+2+2 (both are the same)

Design of child node IDs included in Status

Value Node ID: 1 (01)

LN Node ID: 2 (10)

RN Node ID: 3 (11)

Continue to discuss the following states for any node:

1. Only one Value node is included.

2. Only 2 child nodes are included. Possible states: V+LN or V+RN.

3. Simultaneously including 3 child nodes.

Status bit partition design:

Child node existence flag (3 bits, mandatory), child node size relationship flag (2 bits, optional), child node sorting flag (4 bits, optional).

Analysis and optimization of the above flag positions.

1. Analysis and optimization of child node existence flag bits (3 bits, mandatory).

Based on an ID based MPT tree, if Passport and Transaction are stored, each ID must have a corresponding real object present, so its Value will not be empty. Therefore, the existence flag of the child node only needs to record whether LN and RN exist, so only 2 bits are needed to meet

the requirement of recording the existence of the child node.

In summary, the conclusion after analysis and optimization is as follows:

Child node existence flag bit (2 bits, mandatory).

The first bit records the existence of LN: 0-LN does not exist, 1-LN exists.

The second bit records the existence of RN: 0-RN does not exist, 1-RN exists.

2. Analysis and optimization of child node size relationship flag bits (2 bits, optional)

00,01,10,11

2.1 This flag is not required when there is only one node present.

At this point, its value is 00. At this point, the lowest four digits of the Status must be 0000. At this point, only the nonce of the Value and the difference between the current height need to be stored in the Node.

When there are 2 child nodes:

What status does it need to record: same nonce (01)? Nonce is different (10)?

When there are 3 child nodes:

The status that needs to be recorded: all three nonces are

the same (01)? Are all three nonces different (10)? Are there two similarities among the three nonces (11)?

3. Child node sorting flag bit (4 bits, optional) analysis and optimization

This flag is not required when only the Value node exists.

When there are 2 child nodes:

These two child nodes are the same. Nothing is recorded, so the value is 0000.

These two child nodes are different. Record the ID of the smallest node.

When there are 3 child nodes:

These three child nodes are all the same. Nothing is recorded, so the value is 0000.

These three child nodes are all different. Record the ID of the smallest node and the ID of the largest node.

Two of these three child nodes are the same. Record the ID of the smallest node.

2.11 Trusted State Object Verification Protocol

Trusted State Object Verification Protocol is used to prove that a specific state object has been verified by the full node based on the merkle patricia trie proof associated with it.

2.12 About EQcoinBlock

EQcoinBlock has a EQcoinBlockHeader, and a body (the relevant transactions that are added to the EQcoin blockchain).

2.12.1 About SingularityBlock

The SingularityBlock is the first block of EQcoin, and the No. 1 to No. 1001 Passports will be issued in this block. Due to "Without time at the moment," SingularityBlock timestamp is zero.

2.12.2 About EQcoinBlockHeader

EQcoinBlockHeader has a ProtocolVersion, a Status, a ParentHash, a BlockNumber, a Timestamp, a TotalSupply, a TotalPassportNumbers, a TotalTransactionNumbers , a StateRoot, a TransactionRoot, a PowerUsed, a BeneficiaryPassportID, a Difficulty, and a Nonce.

2.12.3 EQcoinBlock included transactions sorting priority design

EQcoinBlock first packages the transactions with a higher power price (sorted by power price from highest to lowest,

and when the power prices are equal, sorted by Passport ID from smallest to largest) and then packages the transactions with the default power price (sorted by Passport ID from smallest to largest).

Each EQcoinBlock can only include one transaction for a specific Passport, thus maintaining a balance that increases the accounting probability of high-ID Passport transactions.

2.12.4 EQcoinBlock included transactions packaging design

In order to quickly parse EQcoinBlock, the transactions it contains will be divided into N (positive integer) EQcoinBlock pieces of approximately the same size.

The EQcoinBlock piece storage structure consists of two state objects: L(length) state object(the EQcoinBlock piece length, its data type is EQCBits，its count index does not start from 0 but from 69 (because the size of the smallest transaction is 69 bytes)) and V(value) state object(the EQcoinBlock piece body).

This feature is not supported in the Inception phase.

2.12.5 EQcoinBlock's block time interval and maximum TPS

EQcoin's block time interval is about 15 second, and the TPS is approximately 1000.

2.12.6 EQcoinBlock included transactions concurrent execution design

The transaction processing process included in EQcoinBlock consists of two stages: transaction verification stage and transaction execution stage.

2.12.6.1 Transaction verification stage

The transaction verification phase will concurrently verify the validity of all transactions contained in the current block according to the EQcoin transaction verification specifications, such as verifying whether their ordering, signature, nonce, balance, etc. are valid.

2.12.6.2 Transaction execution stage

The transaction execution phase will be executed concurrently in units of execution unit queues. The sub-transaction state objects contained in the transaction will

be placed in the corresponding execution unit queue according to their types. Execution unit queues include Transfer&OP execution unit queues and SmartContract execution unit queues.

The Transfer&OP state objects of each transaction included in the Transfer&OP execution unit queue will be executed concurrently.

The SmartContract execution unit queue is divided into atomic SmartContract execution unit queue and composite SmartContract execution unit queue according to whether there is a coupling relationship between SmartContracts to call each other's smart contract functions.

If a SmartContract does not have a smart contract function calling coupling relationship with any other SmartContract, then it is an atomic SmartContract execution unit queue, and the sub-SmartContract state objects of each transaction it contains will be executed in sequence according to the order in which the transactions are sorted.

If there is a coupling relationship between smart contract functions calling each other between multiple SmartContracts, then they will form a composite SmartContract execution unit queue as a whole, and the sub-SmartContract state objects of

each transaction contained in it will be executed in sequence according to the order in which the transactions are sorted.

The Transfer&OP execution unit queue, each atomic SmartContract execution unit queue, and each composite SmartContract execution unit queue will execute concurrently.

2.13 About EQcoinFBI

EQcoinFBI is an abbreviation for "EQcoin Federated Byzantine Intelligence". It is an intelligent federated Byzantine agreement. It is the consensus mechanism of EQcoin.

2.14 EQcoin roadmap

Stage 1 - Inception

EQcoin mainnet online.

Stage 2 - Era

EQcoin supports Intelligent and EQCVM, decentralized exchange based on Matchmaking Transaction Protocol and Lightning Network (LN).

Stage 3 - New dawn

EQcoin supports cross chain through the Interchain Communication Protocol.

Stage 4 - Nirvana

EQcoin moves from PoW to PoS.

2.15 EQcoin milestones

2018-01-01 EQcoin officially launched.

2018-04-10 GitHub Initial commit.

2019 Establish an EQcoin test network to achieve multiple miner nodes based on PoW consensus mechanisms to mine, send transactions(issue Passport, transfer EQC and change lock), verify blocks, and compete the longest blockchain.

2020-02-10 Register the domain name of www.eqcoin.org.

2021-02-12 Create EQcoin organization in GitHub.

2021-04 Create EQcoin twitter.

2021-04 Create EQcoin facebok.

At present, the overall design of the Inception phase of EQcoin has been completed. We have written thousands of pages of research and development technology documents and the code is about 80% complete and including a total of 33000+ lines.

2.16 EQcoin GitHub

https://github.com/EQcoin

2.17 EQcoin developer community

Currently, we have 12 members and 8 outside collaborators. You can visit https://github.com/orgs/EQcoin/people to learn more about our developer community.

2.18 EQcoin Thanksgiving Day

September 19th is EQcoin Thanksgiving Day.

To express gratitude for the contributions of Bitcoin's founder, Satoshi Nakamoto, a donation of 1,010,000 EQC is being presented to him.

To express gratitude for the contributions of Ethereum's founders - Vitalik Buterin, Anthony Di Iorio, Charles Hoskinson, Mihai Alisie, Amir Chetrit, Joseph Lubin, Gavin Wood, and Jeffrey Wilcke - a donation of 80,000 EQC is being presented to them.

To express gratitude for the contributions of the EQcoin

developer community, a donation of 1,180,000 EQC is being presented to them.

2.19 Copyright

The copyright of all works released by Xun Wang or jointly released by Xun Wang with cooperative partners are owned by Xun Wang and entitled to protection available from copyright law by country as well as international conventions.

Attribution — You must give appropriate credit, provide a link to the license.

Non Commercial — You may not use the material for commercial purposes.

No Derivatives — If you remix, transform, or build upon the material, you may not distribute the modified material.

Xun Wang reserves any and all current and future rights, titles and interests in any and all intellectual property rights of Xun Wang including but not limited to discoveries, ideas, marks, concepts, methods, formulas, processes, codes, software, inventions, compositions, techniques, information and data, whether or not protectable in trademark, copyrightable or patentable, and any trademarks, copyrights or patents based thereon.