

This member-only story is on us. [Upgrade](#) to access all of Medium.

✦ Member-only story

Ethereum Merkle Patricia Trie Explained



Leo Zhang · [Follow](#)

11 min read · Aug 12, 2020



Listen



Share



More

Introduction

Merkle Patricia Trie is one of the key data structures for the Ethereum's storage layer. I wanted to understand how exactly it works. So I did a deep research on all the materials I could find and implemented the algorithm myself.

In this block post, I'll share what I have learned. Explain how exactly Merkle Patricia Trie works, and show you a demo of how a merkle proof is generated and verified.

The source code of the algorithm and the examples used in this blog post are all open sourced.

zhangchiqing/merkle-patricia-trie

This is a simplified implementation of Ethereum's modified Merkle Patricia Trie based on the Ethereum's yellow paper...

[github.com](#)

OK, let's get started.

A basic key-value mapping

Ethereum's Merkle Patricia Trie is essentially a key-value mapping that provides the following standard methods:

```

type Trie interface {
    // methods as a basic key-value mapping
    Get(key []byte) ([]byte, bool) {
    Put(key []byte, value []byte)
    Del(key []byte, value []byte) bool
}

```

An implementation of the above Trie interface should pass the following test cases:

```

func TestGetPut(t *testing.T) {
    t.Run("should get nothing if key does not exist", func(t
    *testing.T) {
        trie := NewTrie()
        _, found := trie.Get([]byte("notexist"))
        require.Equal(t, false, found)
    })

    t.Run("should get value if key exist", func(t *testing.T) {
        trie := NewTrie()
        trie.Put([]byte{1, 2, 3, 4}, []byte("hello"))
        val, found := trie.Get([]byte{1, 2, 3, 4})
        require.Equal(t, true, found)
    })
}

```

Open in app ↗

Medium

Search



```

        trie.Put([]byte{1, 2, 3, 4}, []byte("hello"))
        trie.Put([]byte{1, 2, 3, 4}, []byte("world"))
        val, found := trie.Get([]byte{1, 2, 3, 4})
        require.Equal(t, true, found)
        require.Equal(t, val, []byte("world"))
    })
}

```

(Test cases in this tutorial are included [in the repo](#) and passed.)

Verify Data Integrity

What is merkle patricia trie different from a standard mapping?

Well, merkle patricia trie allows us to verify data integrity. (For the rest of this tutorial, we will call it trie for simplicity)

One can compute the Merkle Root Hash of the trie with the `Hash` function, such that if any key-value pair was updated, the merkle root hash of the trie would be

different; if two Tries have the identical key-value pairs, they should have the same merkle root hash.

```
type Trie interface {
    // compute the merkle root hash for verifying data integrity
    Hash() []byte
}
```

Let's explain this behavior with some test cases:

```
// verify data integrity
func TestDataIntegrity(t *testing.T) {
    t.Run("should get a different hash if a new key-value pair
was added or updated", func(t *testing.T) {
        trie := NewTrie()
        hash0 := trie.Hash()

        trie.Put([]byte{1, 2, 3, 4}, []byte("hello"))
        hash1 := trie.Hash()

        trie.Put([]byte{1, 2}, []byte("world"))
        hash2 := trie.Hash()

        trie.Put([]byte{1, 2}, []byte("trie"))
        hash3 := trie.Hash()

        require.NotEqual(t, hash0, hash1)
        require.NotEqual(t, hash1, hash2)
        require.NotEqual(t, hash2, hash3)
    })

    t.Run("should get the same hash if two tries have the
identical key-value pairs", func(t *testing.T) {
        trie1 := NewTrie()
        trie1.Put([]byte{1, 2, 3, 4}, []byte("hello"))
        trie1.Put([]byte{1, 2}, []byte("world"))

        trie2 := NewTrie()
        trie2.Put([]byte{1, 2, 3, 4}, []byte("hello"))
        trie2.Put([]byte{1, 2}, []byte("world"))

        require.Equal(t, trie1.Hash(), trie2.Hash())
    })
}
```

Verify the inclusion of a key-value pair

Yes, the trie can verify data integrity, but why not simply comparing the hash by hashing the entire list of key-value pairs, why bother creating a trie data structure?

That's because trie also allows us to verify the inclusion of a key-value pair without the access to the entire key-value pairs.

That means trie can provide a proof to prove that a certain key-value pair is included in a key-value mapping that produces a certain merkle root hash.

```
type Proof interface {}

type Trie interface {
    // generate a merkle proof for a key-value pair for verifying the
    // inclusion of the key-value pair
    Prove(key []byte) (Proof, bool)
}

// verify the proof for the given key with the given merkle root
// hash
func VerifyProof(rootHash []byte, key []byte, proof Proof) (value
[]byte, err error)
```

This is useful in Ethereum. For instance, imagine the Ethereum world state is a key-value mapping, and the keys are each account address, and the values are the balances for each account.

As a light client, which don't have the access to the full blockchain state like full nodes do, but only the merkle root hash for certain block, how can it trust the result of its account balance returned from a full node?

The answer is, a full node can provide a merkle proof which contains the merkle root hash, the account key and its balance value, as well as other data. This merkle proof allows a light client to verify the correctness by its own without having access to the full blockchain state.

Let's explain this behavior with test cases:

```
func TestProveAndVerifyProof(t *testing.T) {
    t.Run("should not generate proof for non-exist key", func(t
    *testing.T) {
        tr := NewTrie()
        tr.Put([]byte{1, 2, 3}, []byte("hello"))
```

```

tr.Put([]byte{1, 2, 3, 4, 5}, []byte("world"))
notExistKey := []byte{1, 2, 3, 4}
_, ok := tr.Prove(notExistKey)
require.False(t, ok)
})

t.Run("should generate a proof for an existing key, the
proof can be verified with the merkle root hash", func(t *testing.T)
{
    tr := NewTrie()
    tr.Put([]byte{1, 2, 3}, []byte("hello"))
    tr.Put([]byte{1, 2, 3, 4, 5}, []byte("world"))

    key := []byte{1, 2, 3}
    proof, ok := tr.Prove(key)
    require.True(t, ok)

    rootHash := tr.Hash()

    // verify the proof with the root hash, the key in
question and its proof
    val, err := VerifyProof(rootHash, key, proof)
    require.NoError(t, err)

    // when the verification has passed, it should
return the correct value for the key
    require.Equal(t, []byte("hello"), val)
})

t.Run("should fail the verification if the trie was
updated", func(t *testing.T) {
    tr := NewTrie()
    tr.Put([]byte{1, 2, 3}, []byte("hello"))
    tr.Put([]byte{1, 2, 3, 4, 5}, []byte("world"))

    // the hash was taken before the trie was updated
    rootHash := tr.Hash()

    // the proof was generated after the trie was
updated
    tr.Put([]byte{5, 6, 7}, []byte("trie"))
    key := []byte{1, 2, 3}
    proof, ok := tr.Prove(key)
    require.True(t, ok)

    // should fail the verification since the merkle
root hash doesn't match
    _, err := VerifyProof(rootHash, key, proof)
    require.Error(t, err)
})
}

```

A light client can ask for a merkle root hash of the trie state, and use it to verify the balance of its account. If the trie was updated, even if the updates was to other keys, then the verification would fail.

And now, the light client only needs to trust the merkle root hash, which is a small piece of data, to convince themselves whether the full node returned the correct balance for its account.

OK, but why should the light client trust the merkle root hash?

Since Ethereum's consensus mechanism is Proof of Work, and the merkle root hash for the world state is included in each block head, the computation work is the proof for verifying/trusting the merkle root hash.

It's pretty cool that small as the merkle root hash can be used to verify the state of a giant key-value mapping.

Verify the implementation

I've explained how merkle patricia trie works. [This repo](#) provides a simple implementation. But, how can we verify our implementation?

An easy way is to verify with the Ethereum mainnet data and the official Trie golang implementation.

Ethereum has 3 Merkle Patricia Tries: Transaction Trie, Receipt Trie and State Trie. In each block header, it includes the 3 merkle root hashes: `transactionRoot`, `receiptRoot` and the `stateRoot`.

Since the `transactionRoot` is the merkle root hash of all the transactions included in the block, we could verify our implementation by taking all the transactions, then store them in our trie, compute its merkle root hash, and in the end compare it with the `transactionRoot` in the block header.

For instance, I picked the [block 10467135 on mainnet](#), and saved all the 193 transactions into a [transactions.json](#) file.

Since the transaction root for block 10467135 is [0xbb345e208bda953c908027a45aa443d6cab6b8d2fd64e83ec52f1008ddeafa58](#). I can create a test case that adds the 193 transactions of block 10467135 to our Trie and check:

- Whether the merkle root hash is
bb345e208bda953c908027a45aa443d6cab6b8d2fd64e83ec52f1008ddeafa58 .
- Whether a merkle proof for a certain transaction generated from our trie implementation could be verified by the official implementation.

But what would be the keys and values for the list of transactions? The keys are the RLP encoding of a unsigned integer starting from index 0; the values are the RLP encoding of the corresponding transactions.

OK, let's see the test cases:

```
import (
    "github.com/ethereum/go-ethereum/common"
    "github.com/ethereum/go-ethereum/core/types"
    "github.com/ethereum/go-ethereum/trie"
)

// use the official go lang implementation to check if a valid proof
// from our implementation can be accepted
func VerifyProof(rootHash []byte, key []byte, proof Proof) (value
[]byte, err error) {
    return trie.VerifyProof(common.BytesToHash(rootHash), key,
proof)
}

// load transaction from json
func TransactionJSON(t *testing.T) *types.Transaction {
    jsonFile, err := os.Open("transaction.json")
    defer jsonFile.Close()
    require.NoError(t, err)
    byteValue, err := ioutil.ReadAll(jsonFile)
    require.NoError(t, err)
    var tx types.Transaction
    json.Unmarshal(byteValue, &tx)
    return &tx
}

func TestTransactionRootAndProof(t *testing.T) {
    trie := NewTrie()

    txs := TransactionsJSON(t)

    for i, tx := range txs {
        // key is the encoding of the index as the unsigned
integer type
        key, err := rlp.EncodeToBytes(uint(i))
        require.NoError(t, err)

        transaction := FromEthTransaction(tx)
```

```

        // value is the RLP encoding of a transaction
        rlp, err := transaction.GetRLP()
        require.NoError(t, err)

        trie.Put(key, rlp)
    }

    // the transaction root for block 10467135
    // https://api.etherscan.io/api?module=proxy&action=eth\_getBlockByNumber&tag=0x9fb73f&boolean=true&apikey=YourApiKeyToken
    transactionRoot, err :=
    hex.DecodeString("bb345e208bda953c908027a45aa443d6cab6b8d2fd64e83ec52f1008ddeafa58")
    require.NoError(t, err)

    t.Run("merkle root hash should match with 10467135's transactionRoot", func(t *testing.T) {
        // transaction root should match with block 10467135's transactionRoot
        require.Equal(t, transactionRoot, trie.Hash())
    })

    t.Run("a merkle proof for a certain transaction can be verified by the official trie implementation", func(t *testing.T) {
        key, err := rlp.EncodeToBytes(uint(30))
        require.NoError(t, err)

        proof, found := trie.Prove(key)
        require.Equal(t, true, found)

        txRLP, err := VerifyProof(transactionRoot, key,
proof)
        require.NoError(t, err)

        // verify that if the verification passes, it
returns the RLP encoded transaction
        rlp, err := FromEthTransaction(txs[30]).GetRLP()
        require.NoError(t, err)
        require.Equal(t, rlp, txRLP)
    })
}

```

The above test cases passed, and showed if we add all the 193 transactions of block 10467135 to our trie, then the trie hash is the same as the transactionRoot published in that block. And the merkle proof for the transaction with index 30, generated by our trie, is considered valid by official golang trie implementation.

Merkle Patricia Trie Internal — Trie Nodes

Now, let's take a look at the internal of the trie.

As an example, let's take a look at Block 10593417 on mainnet to show how a transaction trie was built and how is it stored.

```
(03,
f86f826b2585199c82cc0083015f9094e955ede0a3dbf651e2891356ecd0509c1edb
8d9c8801051fdc4efdc0008025a02190f26e70a82d7f66354a13cda79b6af1aa808d
b768a787aeb348d425d7d0b3a06a82bd0518bc9b69dc551e20d772a1b06222edfc5d
39b6973e4f4dc46ed8b196)
```

The value for key `80` is the result of RLP encoding of the first transaction. The value for key `01` is for the second transaction, and so on.

So we will add the above 4 key-value pairs to the trie, and let's see how the internal structure of the trie changes when adding each of them.

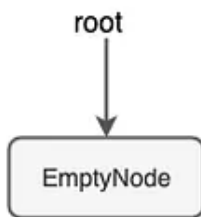
To be more intuitive, I will use some diagrams to explain how it works. You could also inspect the state of each step by adding logs to the test cases.

Empty Trie

The trie structure contains only a root field pointing to a root node. And the Node type is an interface, which could be one of the 4 types of nodes.

```
type Trie struct {
    root Node
}
```

When a trie is created, the root node points to an EmptyNode.

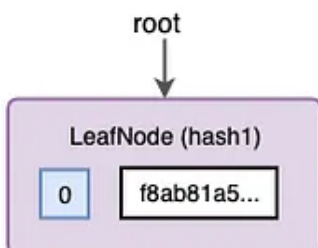


key value pairs

root	keccak(RLP(null))
------	-------------------

Adding the 1st transaction

When adding the key-value pair of the 1st transaction, a LeafNode is created with the transaction data stored in it. And the root node is updated to point to that LeafNode.



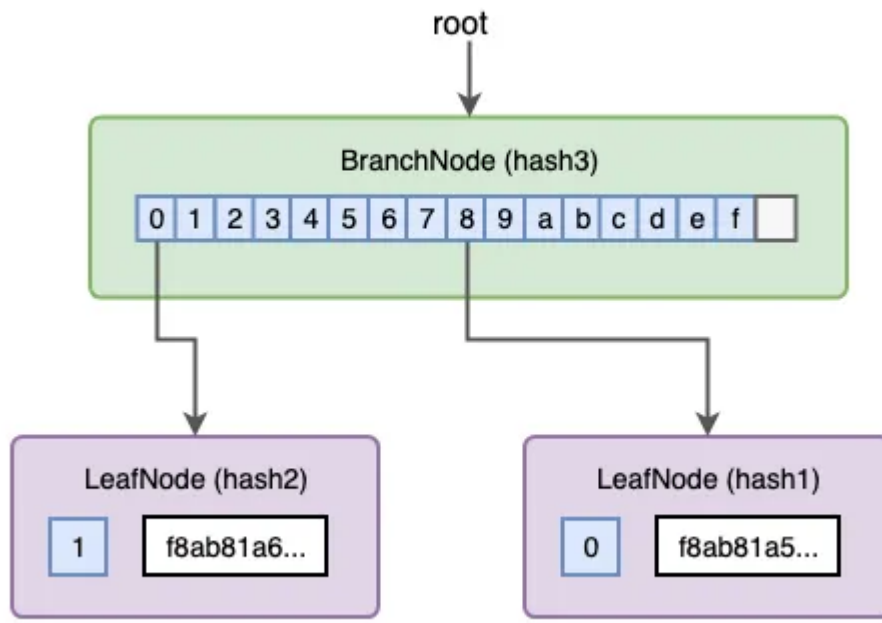
key value pairs

root	hash1
hash1	RLP([80, f8ab81a5...])

Adding the 2nd transaction

When adding the 2nd transaction, the LeafNode at the root will be turned into a BranchNode with two branches pointing to the 2 LeafNodes. The LeafNode on the right side holds the remaining nibbles (nibbles are a single hex character) — 1, and the value for the 2nd transaction.

And now the root node is pointing to the new BranchNode.

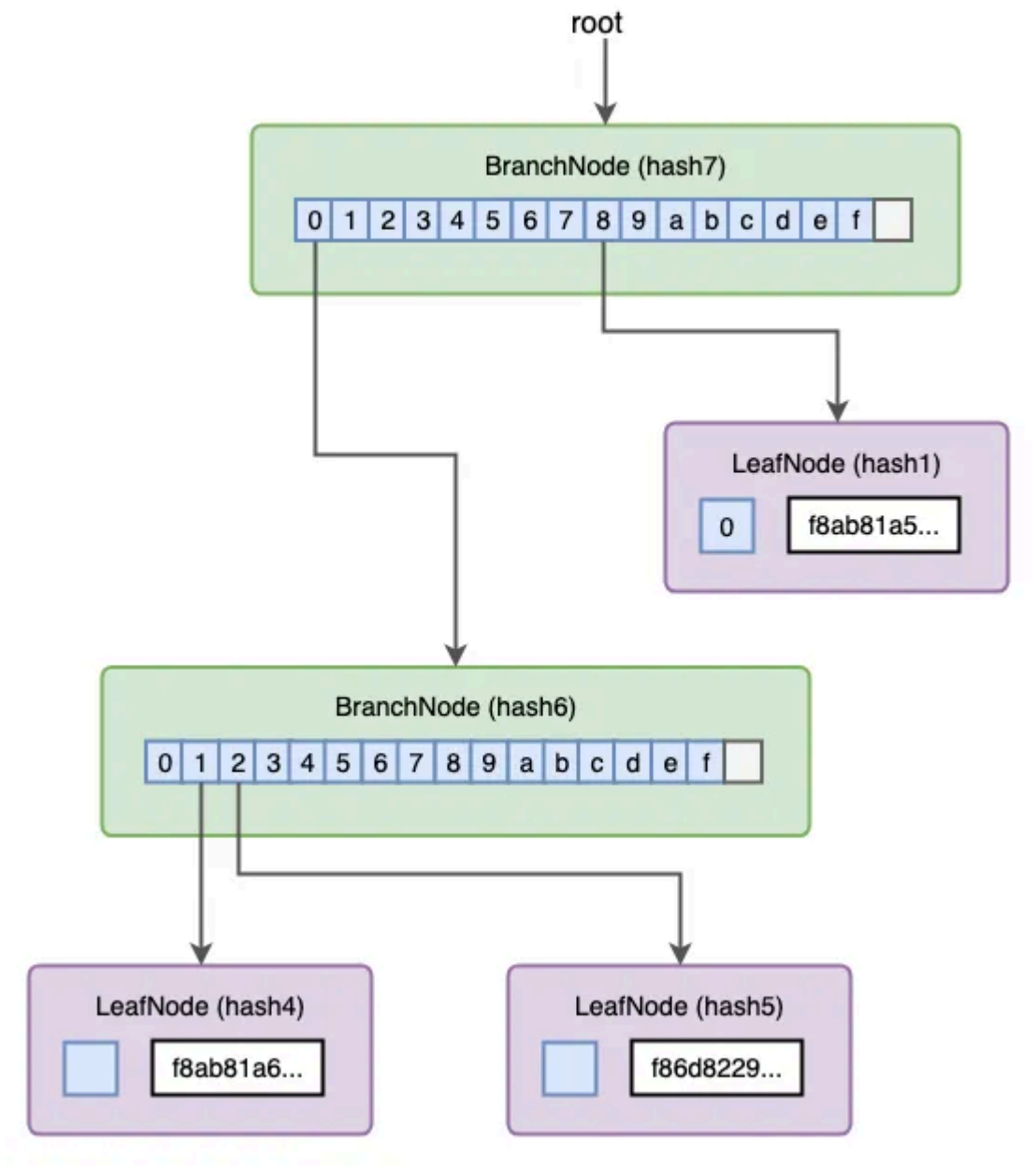


key value pairs

root	hash3
hash3	RLP([hash2, null, null, null, null, null, null, null, hash1, null, null, null, null, null, null, null])
hash2	RLP([1, f8ab81a6...])
hash1	RLP([0, f8ab81a5...])

Adding the 3rd transaction

Adding the 3rd transaction will turn the LeafNode on the left side to be a BranchNode, similar to the process of adding the 2nd transaction. Although the root node didn't change, its root hash has been changed, because it's 0 branch is pointing to a different node with different hashes.

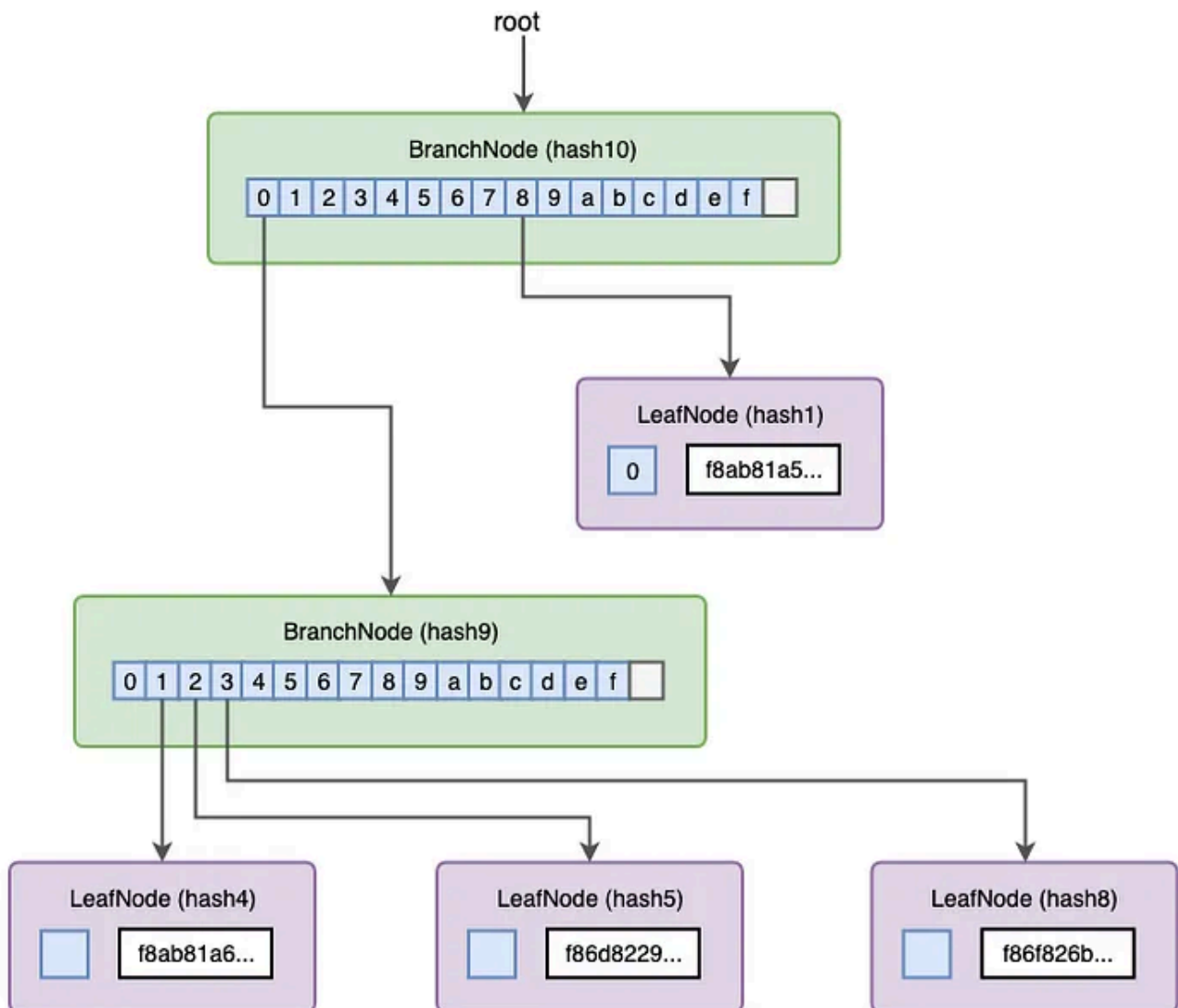


key value pairs

root	hash6
hash7	RLP([hash6, null, null, null, null, null, null, null, hash1, null, null, null, null, null, null, null])
hash6	RLP([null, hash4, hash5, null, null, null, null, null, null, null, null, null, null, null, null, null])
hash5	RLP([, f86d8229...])
hash4	RLP([, f8ab81a6...])
hash1	RLP([0, f8ab81a5...])

Adding the 4th transaction

Adding the last transaction is similar to adding the 3rd transaction. Now we can verify the root hash is identical to the transactionRoot included in the block.



key value pairs

root	hash10
hash10	RLP([hash9, null, null, null, null, null, null, null, hash1, null, null, null, null, null, null, null])
hash9	RLP([null, hash4, hash5, hash8, null, null, null, null, null, null, null, null, null, null])
hash8	RLP([, f86f826b...])
hash5	RLP([, f86d8229...])
hash4	RLP([, f8ab81a6...])
hash1	RLP([0, f8ab81a5...])

Getting Merkle Proof for the 3rd transaction

The Merkle Proof for the 3rd transaction is simply the path to the LeafNode that stores the value of the 3rd transaction. When verifying the proof, one can start from the root hash, decode the Node, match the nibbles, and repeat until find the Node that matches all the remaining nibbles. If found, then the value is the one paired with the key; if not found, then the merkle proof is invalid.

The rule of updating the trie

In the above example, we've built a trie with 3 types of Nodes: EmptyNode, LeafNode and BranchNode. However, we didn't have the chance to use ExtensionNode. Please find other test cases that use the ExtensionNode.

In general, the rule is:

- When stopped at an EmptyNode, replace it with a new LeafNode with the remaining path.
- When stopped at a LeafNode, convert it to an ExtensionNode and add a new branch and a new LeafNode.
- When stopped at an ExtensionNode, convert it to another ExtensionNode with shorter path and create a new BranchNode points to the ExtensionNode.

There are quite some details, if you are interested, you can read the [source code](#).

Summary

Merkle Patricia Trie is a data structure that stores key-value pairs, just like a map. In addition to that, it also allows us to verify data integrity and the inclusion of a key-value pair.

More

Merkle Trie is heavily used in Ethereum storage, if you are interested in learning more, check out my blog post series:

- [Merkle Patricia Trie Explained](#)
- [Verify Ethereum Account Balance with State Proof](#)
- [EIP 1186 — the standard for getting account proof](#)
- [Verify Ethereum Smart Contract State with Proof](#)
- [Verify USDC Balance and NFT Meta Data with Proof](#)

[Merkle Patricia Trie](#)[Blockchain](#)[Merkle Tree](#)[Web3](#)[Ethereum](#)[Follow](#)

Written by Leo Zhang

595 Followers · 92 Following

Flow Builder <https://www.onflow.org/>

Responses (7)



What are your thoughts?

Respond



Mark Odayan
over 2 years ago



keys are the RLP encoding of a unsigned integer starting from index 0; the values are the RLP encoding of the corresponding transactions.

So to clarify:

- node path is the transaction hash
- leaf node has key named by RLP[transactionIndex] with value of RLP encoding of the required transaction tuple

That correct?



Reply



Mark Odayan
over 2 years ago



Whether the merkle root hash is
bb345e208bda953c908027a45aa443d6cab6b8d2fd64e83ec52f1008ddeafa58

I would have referred to this here as the 'transactionRoot' to clarify which specific merkle root hash of the block this belongs to.



Reply



fakedev9999
over 2 years ago



the right side

the left side?



Reply

[See all responses](#)

More from Leo Zhang

			liquidity with 50 tokenA and 20 tokenB	liquidity with 50 tokenA and 200 tokenB
Lisa owns	TokenA	5	0	0
	TokenB	20	0	0
	TokenP	0	10	10
Lily owns	TokenA	50	50	0
	TokenB	200	200	0
	TokenP	0	0	100
	TokenA	0	5	55



Leo Zhang

Uniswap V2 Explained (Beginner Friendly)

Among those keywords about the core tech and mechanisms in crypto mentioned by Vitalik, what is $x*y=k$? It looks quite simple. Isn't it?



Feb 28, 2022

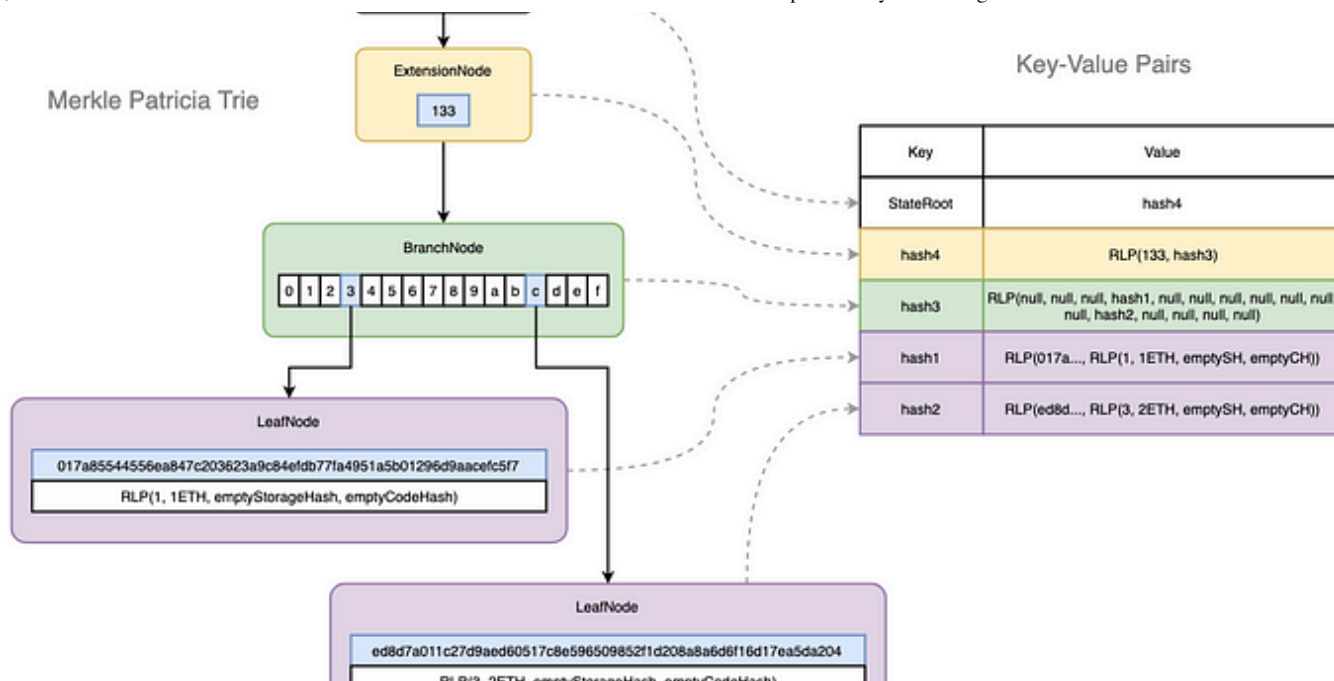


531



5






 Leo Zhang

Verify Ethereum Account Balance with State Proof

How to check the balance of your Ethereum account?

★ Jun 19, 2022 🖱️ 428 💬 3

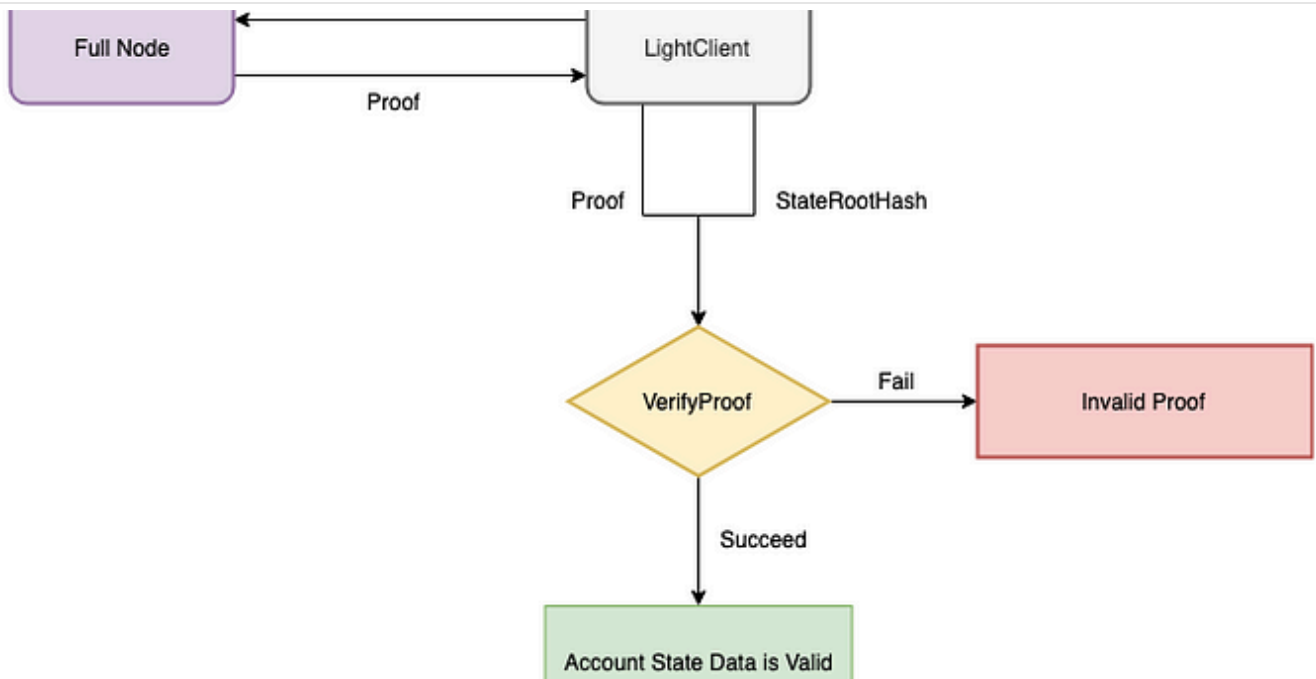


 Leo Zhang

Ethereum Standard ERC165 Explained

In smart contract development, a standard called ERC165 often appears when contract to contract interaction is needed. Solidity already...

★ Oct 20, 2019 🖱 639 💬 5



Leo Zhang

EIP 1186 Explained—the standard for Getting Account Proof


In previous blog post, we introduced how to verify account balance using state proof. But we didn't talk about how to get the proof and...

★ Jun 25, 2022 🖱 340 💬 1

[See all from Leo Zhang](#)

Recommended from Medium



 Muhammad Ihsan

How Blockchain Works.

In this article, we will discuss the components and mechanisms within blockchain that make it secure.

✦ Oct 4, 2024 🖱 5 💬 1

🔖+ ⋮



 Swastika Yadav

Merkle proofs explained

With the increasing adoption of blockchain technology, Merkle proofs have become crucial for ensuring data integrity and efficiency. They...

Jul 18, 2024 🖱 61



Lists



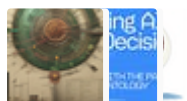
My Kind Of Medium (All-Time Faves)

106 stories · 629 saves



MODERN MARKETING

204 stories · 975 saves



data science and AI

40 stories · 311 saves



Generative AI Recommended Reading

52 stories · 1587 saves



Kashish Gupta

My Interview Journey at Rippling: A Detailed Account with Code Insights

Interviewing for a role at Rippling was an experience that tested my technical skills, design thinking, and ability to handle complex...



Sep 1, 2024



14



Forward Pass Equation

$$\text{Var}[y_k] = \text{Var}\left[\sum_{i=1}^{n_k} x_k^i W_k^{ij} + b_k^j\right] \Rightarrow \text{Var}[W_k] = \frac{2}{n_k}$$

Backward Pass Equation

$$\text{Var}[\Delta y_k] = \text{Var}[\Delta x_{k+1} f'(y_k)] \Rightarrow \text{Var}[W_k] = \frac{2}{n_k}$$

Weight Distributions

$$\text{Var}[W_k] = \frac{2}{n_k} \Rightarrow \begin{cases} W_k \sim N(0, \sigma^2) \Rightarrow \sigma = \sqrt{\frac{2}{n_k}} \\ W_k \sim U(-a, a) \Rightarrow a = \sqrt{\frac{6}{n_k}} \end{cases}$$



In Towards Data Science by Ester Hlav

Kaiming He Initialization in Neural Networks—Math Proof

Deriving optimal initial variance of weight matrices in neural network layers with ReLU activation function



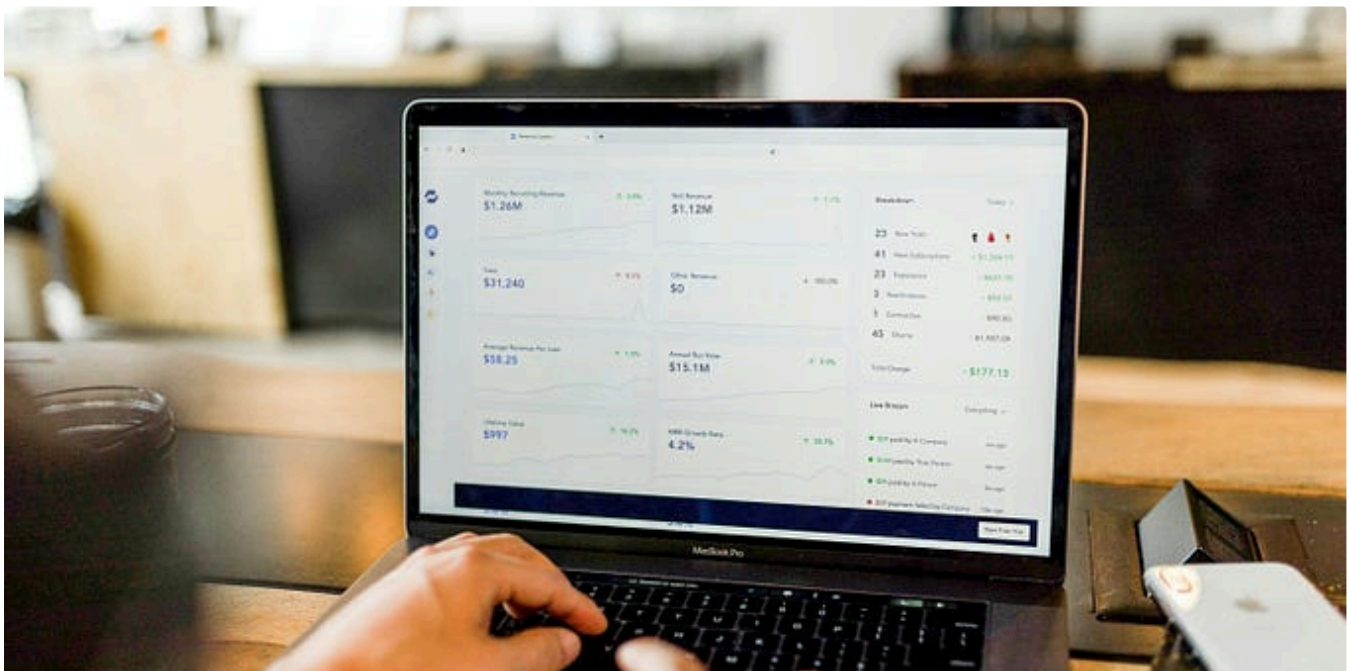
Feb 15, 2023



231



4

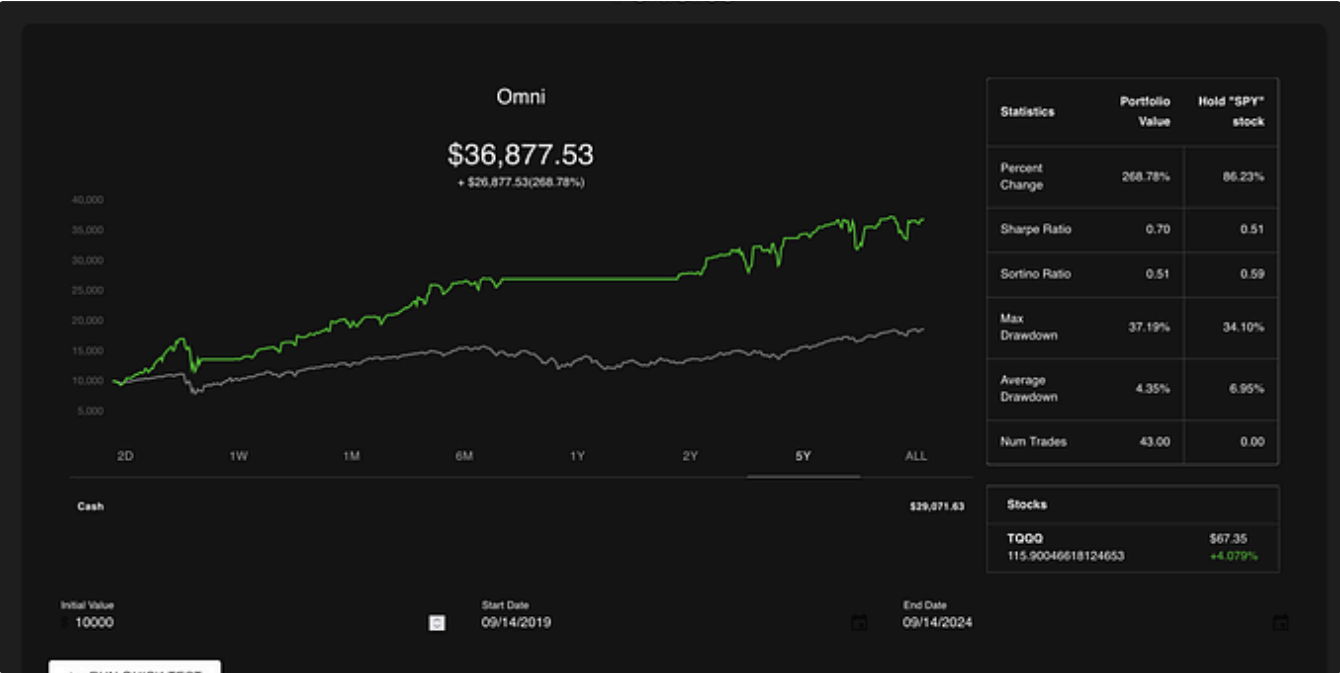
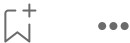


Taofikat Titilayo Adeleke

Ethereum Layer Two Solutions: Scaling Blockchain

Ever since its start, Ethereum has become a key player in the blockchain world. It's the second most valuable platform, next to Bitcoin...

★ Jul 18, 2024



In DataDrivenInvestor by Austin Starks

I used OpenAI's o1 model to develop a trading strategy. It is DESTROYING the market

It literally took one try. I was shocked.

★ Sep 15, 2024 🖱️ 8.1K 💬 205



See more recommendations