智能合约实现白名单的3个机制

原创 翻译小组 登链社区 2022-06-01 18:00 发表于广东

收录于合集
#Solidity 54 #签名 5

> 译文出自：登链翻译计划[1]
>
> 译者：翻译小组[2]
>
> 校对：Tiny 熊[3]

简介

白名单是推广 NFT 项目和奖励早期进入及热情参与者的好方法。有很多方法可以实现白名单机制，每种方法都有自己的优势和劣势。现在主要有 3 种实现白名单机制的方法，本文介绍它们，并谈谈它们的优点和缺点。

最原始的方式--将白名单保存在存储中

对于熟悉其他语言和现代计算系统的开发者来说，将数据存储在堆或存储器中似乎是处理一系列数据的一个相当合理和简单的方法。

因此，要将白名单存储在存储器中，你可以简单地声明一个 mapping 映射，记录所有符合白名单条件的有效地址。然而，在 EVM 中，使用这种方式将消耗你**大量的 Gas**，而且是一种非常低效的方法。不过，对于任何人来说，通过 etherscan 或几行代码来测试他或她是否在白名单上将会更容易。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract PrimitiveWhiteList is ERC721Enumerable, Ownable {

  uint256 public constant MINT_PRICE = 0.1 ether;
  mapping(address => bool) public whitelist;
```

```
  constructor() ERC721("Primitive Whitelist", "PW") {}

  function whitelistMint(uint256 amount) external payable {
    require(msg.value == amount * MINT_PRICE, "Ether send below price");
    require(whitelist[msg.sender], "Not in whitelist");

    // start minting
    uint256 currentSupply = totalSupply();

    for (uint256 i = 1; i <= amount; i++) {
        _safeMint(msg.sender, currentSupply + i);
    }
  }

  function addWhitelist(address _newEntry) external onlyOwner {
    whitelist[_newEntry] = true;
  }

  function removeWhitelist(address _newEntry) external onlyOwner {
    require(whitelist[_newEntry], "Previous not in whitelist");
    whitelist[_newEntry] = false;
  }

}
```

优点：容易验证，编码简单，容易添加地址或删除地址

缺点：效率真的很低，对于发布者（项目方）来说真的很贵

聪明的方法 I - Merkle Tree Airdrop

执行白名单的另一种方式是利用默克尔(Merkle)树。Merkle 树是区块链中的一个重要角色。Merkle 树利用了 hash 的特性，即输入的轻微变化将导致完全不同的输出，以及两个输入导致相同输出的概率几乎是不可能的事实。

merkle 树的结构如下所示。当前节点的哈希值等于其左侧子节点、右侧子节点及其数据的哈希值。因此，从散列的属性来看，任何变化都会导致完全不同的输出；我们可以利用这个属性来实现白名单。

所以，让我们想象一下，你想知道 L1 是否等于一个地址，你可以提取 Hash 0-1，Hash 1，以及被测试的地址。然后，你按照 hash 规则，将 hash 的输出与 Top Hash 进行比较。如果结果相同，你可以确保 L1 等于输入地址。

然而，要做到这一点，你需要生成一棵 merkle 树，并将一半的构建树过程从链上拿走，以节省 Gas。我们将使用 javascript 来生成 merkle 树。如果你对 ether.js 比较熟悉，你也可以使用*ethers.utils.solidityKeccak256*来哈希配对。另外，记得要将 JSON 文件存储为以下内容：{ <address> :, <address> :}。你也可以根据你的需要，把数据改成另一种类型来调整。

```
import fs from "fs";
import { MerkleTree } from "merkletreejs";
import Web3 from "web3";
import keccak256 from "keccak256";
import dotenv from "dotenv";

// Establish web3 provider
dotenv.config();
const web3 = new Web3(process.env.MAINNET_RPC_URL);

// hashing function for solidity keccak256
const hashNode = (account, amount) => {
    return Buffer.from(
        web3.utils
            .soliditySha3(
                { t: "address", v: account },
                { t: "uint256", v: amount }
            )
```

```
            .slice(2),
        "hex"
    );
};

// read list, Note: the root path is at cwd
// the json file structure: {"<address>": <amount>, "<address>": <amount>, ...}
const readRawList = (path) => {
    const rawdata = fs.readFileSync(path);
    const data = JSON.parse(rawdata);

    return data;
};

const generateMerkleTree = (data) => {
    const leaves = Object.entries(data).map((node) => hashNode(...node));

    const merkleTree = new MerkleTree(leaves, keccak256, { sortPairs: true });
    const merkleRoot = merkleTree.getHexRoot();

    return [merkleRoot, merkleTree];
};

const checkTree = (pairs, tree, root) => {
    for (const [key, value] of Object.entries(pairs)) {
        const leaf = hashNode(key, value);
        const proof = tree.getProof(leaf);

        // hex proof for solidity byte32[] input
        // const hexProof = tree.getHexProof(leaf);

        if (!tree.verify(proof, leaf, root)) {
            console.err("Verification failed");
            return false;
        }
    }

    return true;
};

function main(filepath, outputPath) {
    const rawData = readRawList(filepath);
    const [merkleRoot, merkleTree] = generateMerkleTree(rawData);

    if (checkTree(rawData, merkleTree, merkleRoot)) {
```

```
    if (checkTree(rawData, merkleTree, merkleRoot)) {
        fs.writeFileSync(
            outputPath,
            JSON.stringify({
                root: merkleRoot,
                tree: merkleTree,
            })
        );

        console.log(`Successfully generate merkle tree to ${outputPath}.`);
    } else {
        console.err("Generate merkle tree failed.");
    }
}

main("./db/freeClaimList.json", "./db/freeClaimMerkle.json");
```

我们需要通过 solidity 函数将 hash 根存储在合约中。然后用库 MerkleProof 进行链上验证。因此，每当有人想铸币时，你必须为用户生成证明，无论是在前端还是后端，用 tree.getHexProof 函数生成 bytes32[]证明。你可以查看 checkTree 函数中的注释，了解更详细的实现。

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
import "@openzeppelin/contracts/utils/cryptography/MerkleProof.sol";

contract PrimitiveWhiteList is ERC721Enumerable, Ownable {

  using ECDSA for bytes32;

  uint256 public constant MINT_PRICE = 0.1 ether;
  bytes32 private _whitelistMerkleRoot;

  constructor() ERC721("Merkle Tree Whitelist", "MTW") {}

  function whitelistSale(bytes32[] memory proof, uint256 amount) external payable {
        // merkle tree list related
        require(_whitelistMerkleRoot != "", "Free Claim merkle tree not set");
        require(
            MerkleProof.verify(
                proof,
                _whitelistMerkleRoot,
                keccak256(abi.encodePacked(msg.sender, amount))
            ),
            "Free Claim validation failed"
        );
```

```
        );

        // start minting
        uint256 currentSupply = totalSupply();

        for (uint256 i = 1; i <= amount; i++) {
            _safeMint(msg.sender, currentSupply + i);
        }
    }

    function setWhitelistMerkleRoot(bytes32 newMerkleRoot_) external onlyOwner {
        _whitelistMerkleRoot = newMerkleRoot_;
    }

}
```

然而，这意味着每当我们想调整白名单时，必须更新_freeClaimMerkleRoot。

优点：经济效益高（对项目方来说），易于验证

缺点：对用户来说，铸造 gas 成本的略大，以及每次希望改变白名单时，都需要重新设置 hash 根。

聪明的方法二 - 后台签名

最后一种方式也比第一种方式更便宜。然而，最后一种方式比前一种方式更中心化一些。这种方法利用了签名机制。你需要在后端设置一个地址，并确保它的可信度。然后，每当一个白名单上的用户希望铸造时，你首先需要在后端验证。验证后，后端将签名信息传回给用户。之后，用户就可以使用签名信息传给合约进行铸币。

但你可能会问，如何确保没有人可以伪造签名信息？原因是，使用私钥签署信息后会得到一个 hash 信息。然后，你可以用 hash 信息生成公钥地址。因此，在合约处存储公钥地址，在后端用私钥签署消息，就可以确保没有人可以伪造消息。

然而，为了防止重放攻击，你可以使用一个 nonce 来确保签署的消息不会被恶意使用。因此，整个签名过程将如下所示：

```
import Web3 from "web3";
import dotenv from "dotenv";

// Establish web3 provider
dotenv.config();
const web3 = new Web3(process.env.MAINNET_RPC_URL);

const generateNonce = () => {
```

```
const generateNonce = () => {
  return crypto.randomBytes(16).toString("hex");
};

// Hash message
const mintMsgHash = (recipient, amount, newNonce, contract) => {
  return (
    web3.utils.soliditySha3(
      { t: "address", v: recipient },
      { t: "uint256", v: amount },
      { t: "string", v: newNonce },
      { t: "address", v: contract }
    ) || ""
  );
};

const signMessage = (msgHash, privateKey) => {
    return web3.eth.accounts.sign(msgHash, privateKey);
};

// Signing the message at backend.
// You can store the data at database or check for Nonce conflict
export const Signing = (address, amount) => {
  const newNonce = generateNonce();

  const hash = mintMsgHash(
    address,
    amount,
    newNonce,
    config.ContractAddress
  );

  const signner = signMessage(hash, config.PrivateKey);

  return {
    amount: amount,
    nonce: newNonce,
    hash: signner.message,
    signature: signner.signature,
  };

}
```

因此，在你生成 hash 消息后，以及相应签名内容打包传递给前端，让用户来铸造 NFT 时，在链上进行验证：

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.11;

import "@openzeppelin/contracts/token/ERC721/ERC721";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

contract SignatureWhitelist is ERC721, Ownable {
  using ECDSA for bytes32;

  address private _systemAddress;
  mapping(string => bool) public _usedNonces;

  function publicSale(
    uint256 amount,
    string memory nonce,
    bytes32 hash,
    bytes memory signature
  ) external payable nonReentrant {

    // signature realted
    require(matchSigner(hash, signature), "Plz mint through website");
    require(!_usedNonces[nonce], "Hash reused");
    require(
      hashTransaction(msg.sender, amount, nonce) == hash,
      "Hash failed"
    );

    _usedNonces[nonce] = true;

    // start minting
    uint256 currentSupply = totalSupply();

    for (uint256 i = 1; i <= amount; i++) {
      _safeMint(msg.sender, currentSupply + i);
    }

  }

  function matchSigner(bytes32 hash, bytes memory signature) public view returns (bool) {
    return _systemAddress == hash.toEthSignedMessageHash().recover(signature);
  }

  function hashTransaction(
    address sender,
    uint256 amount,
    string memory nonce
  ) public view returns (bytes32) {

    bytes32 hash = keccak256(
      abi.encodePacked(sender, amount, nonce, address(this))
    );
```

```
    return hash;
  }

}
```

优点：对开发者来说更便宜，在后台更容易管理白名单。

缺点：用户需要更多的 Gas 来铸币，不那么去中心化

结论

有很多方法可以实现白名单机制。每种方式都有其优点和缺点。因此，开发者应该仔细考虑需求，并在每种方式之间找到余额。另外，我将继续关注这篇文章，并写一篇深入考察这三种方法的 Gas。请关注我的Twitter[4]。

最后但不是最不重要的，我目前正在运行一个 NFT 项目，希望给我们的持有人提供被动收入，可改变的 NFT，项目和 NFT 拍卖之间的联系。加入我们的 Discord，关注我们的 Twitter，并看看我们的网站。

本翻译由 Duet Protocol[5] 赞助支持。

原文：https://coinsbench.com/smart-contract-whitelist-mechanism-fbe3464159ed

参考资料

[1] 登链翻译计划: *https://github.com/lbc-team/Pioneer*

[2] 翻译小组: *https://learnblockchain.cn/people/412*

[3] Tiny 熊: *https://learnblockchain.cn/people/15*

[4] Twitter: *https://twitter.com/hugiRIS_nft*

[5] Duet Protocol: *https://duet.finance/?utm_souce=learnblockchain*

收录于合集 #Solidity 54

下一篇 · Foundry 教程: 用Solidity编写ERC-20测试用例

阅读原文

喜欢此内容的人还喜欢

创建一个基于链上实时数据的动态SVG NFT

登链社区