

仓库是一个基于 *Rust* 语言实现的简单比特币系统，其中包含 *B1*, *B2*, *B3* 三个 *demo*。

## 一. B1 代码树

```
Rust_chain/
├── B1/
│   ├── Cargo.lock
│   ├── Cargo.toml
│   └── src/
│       ├── block.rs
│       ├── blockchain.rs
│       ├── cli.rs
│       ├── merkle_tree.rs
│       ├── node.rs
│       ├── transaction.rs
│       └── main.rs
```

从项目代码模块的角度对 B1 进行分析，B1 项目是一个区块链实现，其代码模块组织较为清晰，各模块负责不同的功能，共同构建了一个基本的区块链系统。以下是对各主要代码模块的详细分析：

---

### 核心模块分析

#### 1. 区块模块 (`block.rs`)

区块是区块链的基本单位，包含以下核心功能：

**区块结构：**定义区块的字段，如索引、时间戳、交易列表、哈希值等。

**哈希计算：**通过 SHA-256 计算区块的哈希值。

**挖矿：**实现工作量证明 (PoW) 算法。

#### 部分核心代码

```
RUST
impl Block {
    // 计算区块的哈希值
    pub fn calculate_hash(&self) -> String {
```

```

        let input = format!(
            "{}{}{}{}{}{}{}",
            self.index,
            self.timestamp,
            self.merkle_root,
            self.transactions.len(),
            self.previous_hash,
            self.nonce
        );
        let mut hasher = Sha256::new();
        hasher.update(input);
        format!("{:x}", hasher.finalize())
    }
    // 挖矿 (PoW)
    pub fn mine_block(&mut self, difficulty: usize) {
        let target = "0".repeat(difficulty);
        while &self.hash[..difficulty] != target {
            self.nonce += 1;
            self.hash = self.calculate_hash();
        }
        println!("Block mined: {}", self.hash);
    }
}

```

## 2. 区块链模块 (blockchain.rs)

区块链模块负责管理区块的链式结构，提供以下功能：

- **创世区块**：创建区块链的第一个区块。
- **添加新区块**：将新区块添加到链中。
- **验证链完整性**：检查区块链的有效性。
- **文件存储**：支持区块链的持久化存储。

部分核心代码

```

RUST
impl Blockchain {
    ... ..
    // 添加新区块
    pub fn add_block(&mut self, transactions: Vec<Transaction>) {
        let latest_block = self.get_latest_block();
        let mut new_block = Block::new(
            latest_block.index + 1,
            Utc::now().timestamp(),

```

```

        transactions,
        latest_block.hash.clone(),
    );
    new_block.mine_block(self.difficulty);
    self.chain.push(new_block);
}
// 验证区块链的完整性
pub fn is_chain_valid(&self) -> bool {
    for i in 1..self.chain.len() {
        let current_block = &self.chain[i];
        let previous_block = &self.chain[i - 1];
        if current_block.hash != current_block.calculate_hash() {
            println!("Invalid hash for block {}", current_block.index);
            return false;
        }
        if current_block.previous_hash != previous_block.hash {
            println!("Invalid previous hash for block {}", current_block.index);
            return false;
        }
    }
    true
}
... ..
}

```

### 3. 交易模块 (transaction.rs)

交易模块定义了区块链中的交易结构，支持以下功能：

- **交易创建**：生成新的交易。
- **签名与验证**：使用 ED25519 算法对交易进行签名和验证。

部分核心代码

```

RUST
impl Transaction {
    // 对交易进行签名
    pub fn sign(&mut self, key_pair: &Ed25519KeyPair) {
        let message = self.to_message();
        let signature_bytes = key_pair.sign(&message).as_ref().to_vec();
        ;
        self.signature = hex::encode(signature_bytes);
    }
}

```

```

    }
    // 验证交易的签名
    pub fn verify(&self) -> bool {
        let message = self.to_message();
        let public_key_bytes = hex::decode(&self.sender).unwrap();
        let signature_bytes = hex::decode(&self.signature).unwrap();
        let public_key = ring::signature::UnparsedPublicKey::new(&ring::signature::ED25519, &public_key_bytes);
        public_key.verify(&message, &signature_bytes).is_ok()
    }
}

```

#### 4. 默克尔树模块 (merkle\_tree.rs)

默克尔树用于高效验证交易数据的完整性，支持以下功能：

- **构建默克尔树**：从交易列表生成默克尔树。
- **计算根哈希**：获取默克尔树的根哈希值。

部分核心代码

```

RUST
impl MerkleTree {
    // 创建一个新的默克尔树
    pub fn new(transactions: Vec<String>) -> Self {
        let mut nodes = transactions
            .into_iter()
            .map(|tx| {
                let hash = Self::hash_leaf(&tx);
                MerkleNode {
                    hash,
                    left: None,
                    right: None,
                }
            })
            .collect::<Vec<_>>();
        while nodes.len() > 1 {
            let mut new_level = Vec::new();
            for chunk in nodes.chunks(2) {
                let left = &chunk[0];
                let right = if chunk.len() > 1 { &chunk[1] } else { &chunk[0] };
                let hash = Self::hash_nodes(&left.hash, &right.hash);
                new_level.push(MerkleNode {

```

```

        hash,
        left: Some(Box::new(left.clone())),
        right: Some(Box::new(right.clone())),
    });
}
nodes = new_level;
}
MerkleTree {
    root: Some(nodes.remove(0)),
}
}
}

```

## 5. 钱包模块 (wallet.rs)

钱包模块用于管理用户的密钥对和地址，支持以下功能：

- **密钥生成**：生成 ED25519 密钥对。
- **地址生成**：从公钥生成钱包地址。
- **签名与验证**：对数据进行签名和验证。

部分核心代码

```

RUST
impl Wallet {
    // 创建一个新钱包
    pub fn new() -> Self {
        let rng = SystemRandom::new();
        let pkcs8_bytes = Ed25519KeyPair::generate_pkcs8(&rng).unwrap();
;
        let key_pair = Ed25519KeyPair::from_pkcs8(pkcs8_bytes.as_ref()).
unwrap();
        Wallet { key_pair }
    }
    // 获取钱包地址
    pub fn address(&self) -> String {
        hex::encode(self.key_pair.public_key().as_ref())
    }
}

```

## 6. CLI 模块 (cli.rs)

CLI 模块提供了命令行交互界面，支持以下功能：

- **创建钱包**：生成新的钱包。
- **发起交易**：创建并签名交易。
- **挖矿**：挖矿新区块。
- **验证区块链**：检查区块链的完整性。

### 部分核心代码

```
RUST
impl Cli {
    pub fn run(&self) {
        match &self.command {
            Commands::CreateWallet => {
                let wallet = Wallet::new();
                println!("New wallet created!");
                println!("Wallet address: {}", wallet.address());
            }
            Commands::AddTransaction { sender, receiver, amount } => {
                let wallet = Wallet::new();
                let transaction = Transaction::new(
                    sender.clone(),
                    receiver.clone(),
                    *amount,
                    &wallet.key_pair,
                );
                println!("Transaction created: {:?}", transaction);
            }
        }
    }
}
```

## 总结

### 模块间的关系和协作

- `blockchain.rs` 模块作为核心模块，管理着整个区块链，依赖于 `block.rs` 模块创建和管理区块，依赖于 `transaction.rs` 模块处理交易。
- `block.rs` 模块负责区块的创建和操作，依赖于 `transaction.rs` 模块处理交易，依赖于 `merkle_tree.rs` 模块计算默克尔根。
- `transaction.rs` 模块负责交易的处理，依赖于 `wallet.rs` 模块进行签名和验证操作。
- `wallet.rs` 模块负责钱包的管理，提供签名和验证功能。
- `pow.rs` 模块实现了工作量证明机制，为 `block.rs` 模块的挖矿操作提供支持。
- `node.rs` 模块负责节点的管理，依赖于 `blockchain.rs` 模块管理区块链。

B1 的代码模块设计清晰，功能完整，适合初学者学习和理解区块链的核心概念。每个模块都有明确的职责，代码注释详细，便于扩展和优化。

## 二. B2 代码树

以下是基于 B2 的代码，从项目代码模块的角度进行的分析，说明 B2 的代码树基于 B1 进行了哪些操作，并解释 `main.rs` 的不同之处：

```
TEXT
src/ |—— block.rs           # 区块核心结构
    |—— blockchain.rs      # 区块链管理
    |—— transaction.rs     # 交易系统
    |—— merkle_tree.rs     # 默克尔树实现
    |—— node.rs           # P2P 网络节点
    |—— pow.rs            # 工作量证明模块
    |—— utils.rs          # 辅助工具集
    |—— main.rs           # 主程序入口
```

### B2 基于 B1 的改动

#### 1. 模块精简

B2 在 B1 的基础上进行了模块精简，移除了以下模块：

- `cli.rs`: B1 中的命令行交互模块被移除，B2 直接通过 `main.rs` 运行示例。
- `wallet.rs`: B1 中的钱包管理模块被移除，B2 直接在 `transaction.rs` 中生成密钥对。

#### 2. 功能优化

B2 对 B1 的部分功能进行了优化：

- 区块结构**：移除了冗余字段，简化了区块的哈希计算。
- 区块链管理**：移除了文件存储功能，专注于内存中的区块链操作。
- 交易系统**：简化了交易签名和验证的逻辑。

#### 3. 关注模块

- 3.1 `node.rs`：实现了 P2P 网络节点的基本功能，支持节点间的区块链同步。

```
impl Node {
    // 创建一个新节点
    pub fn new(address: SocketAddr, difficulty: usize) -> Self {
        Node {
            address,
            blockchain: Arc::new(Mutex::new(Blockchain::new(difficulty))),
        },
        peers: Vec::new(),
    }
}

// 同步区块链
pub fn sync_blockchain(&self) {
    for peer in &self.peers {
        println!("Syncing blockchain with peer: {}", peer);
    }
}
}
```

### 3.2 main.rs 的不同

#### B1 的 main.rs

B1 的 main.rs 通过 CLI 模块与用户交互，支持命令行操作。

RUST

```
fn main() {
    let cli = Cli::parse();
    cli.run();
}
```

#### B2 的 main.rs

B2 的 main.rs 直接运行示例，模拟两个节点间的区块链同步。

RUST

```
fn main() {
    // 生成密钥对
    let alice_key_pair = generate_key_pair();
    let bob_key_pair = generate_key_pair();
    // 创建两个节点
    let node1 = Arc::new(Mutex::new(Node::new(
        SocketAddr::new(IpAddr::from([127, 0, 0, 1]), 8080),
        4, // 难度
    )));
```



```

let node2 = Arc::new(Mutex::new(Node::new(
    SocketAddr::new(IpAddr::from([127, 0, 0, 1]), 8081),
    4, // 难度
)));
// 添加对等节点
node1.lock().unwrap().add_peer(node2.lock().unwrap().address);
node2.lock().unwrap().add_peer(node1.lock().unwrap().address);
// 启动节点
let node1_clone = Arc::clone(&node1);
let node2_clone = Arc::clone(&node2);
let handle1 = thread::spawn(move || {
    let mut node = node1_clone.lock().unwrap();
    // 添加一笔交易
    let transaction = Transaction::new(
        hex::encode(alice_key_pair.public_key().as_ref()), // Alice
        hex::encode(bob_key_pair.public_key().as_ref()), // Bob 的
        100, // 交易金额
        &alice_key_pair, // Alice 的密钥对（用于签名）
    );
    // 将交易添加到新区块
    node.blockchain.lock().unwrap().add_block(vec![transaction]);
    // 同步区块链
    node.sync_blockchain();
    // 打印节点 1 的区块链
    println!("Blockchain on Node 1:");
    for block in &node.blockchain.lock().unwrap().chain {
        println!("{:?}", block);
    }
});
let handle2 = thread::spawn(move || {
    let mut node = node2_clone.lock().unwrap();
    // 同步区块链
    node.sync_blockchain();
    // 打印节点 2 的区块链
    println!("Blockchain on Node 2:");
    for block in &node.blockchain.lock().unwrap().chain {
        println!("{:?}", block);
    }
});
handle1.join().unwrap();
handle2.join().unwrap();
}

```

## 总结

B2 在 B1 的基础上进行了模块精简和功能优化，移除了 CLI 和钱包模块，关注演示网络节点模块，专注于区块链的核心功能和网络同步。`main.rs` 从 CLI 交互改为直接运行示例，模拟两个节点间的区块链同步。

## 三. B3 代码树

```
TEXT
src/
├── block.rs           # 区块核心结构
├── blockchain.rs      # 区块链管理
├── transaction.rs     # 交易系统
├── merkle_tree.rs     # 默克尔树实现
├── node.rs            # P2P 网络节点
├── cli.rs             # 命令行交互界面
├── wallet.rs          # 钱包管理
├── smart_contract.rs  # 智能合约支持
├── privacy.rs         # 隐私交易实现
├── pow.rs             # 工作量证明模块
├── utils.rs           # 辅助工具集
└── main.rs            # 主程序入口
```

### 已拓展的部分

#### 1. 智能合约模块 (`smart_contract.rs`)

B3 新增了智能合约模块，支持合约的部署和执行。

#### 当前状态

**功能：**支持简单的合约部署和执行，提供基础的 `set` 和 `get` 方法。

**问题：**

- 缺乏 Gas 机制，无法防止资源滥用。
- 缺乏事件日志，难以跟踪合约执行状态。
- 安全性不足，容易受到重入攻击等漏洞的影响。

#### 可完善的点

- **Gas 机制：**为合约执行引入 Gas 费用机制。
- **事件日志：**添加合约事件日志功能。
- **安全性：**增强合约执行的安全性，防止重入攻击。

## 代码提示

RUST

```
// 引入 Gas 机制
pub struct GasCounter {
    gas_limit: u64,
    gas_used: u64,
}

impl GasCounter {
    pub fn new(gas_limit: u64) -> Self {
        GasCounter { gas_limit, gas_used: 0 }
    }

    pub fn charge(&mut self, amount: u64) -> Result<(), String> {
        self.gas_used += amount;
        if self.gas_used > self.gas_limit {
            Err("Out of gas".to_string())
        } else {
            Ok(())
        }
    }
}

// 增强安全性
pub fn secure_execute(&mut self, method: &str, args: Vec<String>) -> Result<String, String> {
    if method == "withdraw" {
        // 防止重入攻击
    }
    self.execute(method, args)
}
```

## 2. 隐私交易模块 (privacy.rs)

B3 新增了隐私交易模块，使用零知识证明（ZKP）保护交易隐私。

### 当前状态

- **功能：**支持创建和验证隐私交易的框架。
- **问题：**
  - 框架内各个参数需要进行补充
  - 性能较差，ZKP 的生成和验证开销较大。
  - 缺乏范围证明，无法确保交易金额的合理性。
  - 不支持批量验证，验证效率较低。

## 实现隐私交易

```
RUST
use bellman::groth16::{Proof, VerifyingKey};
use bellman::pairing::bn256::{Bn256, Fr};
use bellman::{Circuit, ConstraintSystem, SynthesisError};
#[derive(Clone)]
struct PrivacyCircuit {
    amount: Option<u64>,
}
impl Circuit<Fr> for PrivacyCircuit {
    fn synthesize<CS: ConstraintSystem<Fr>>(&self, cs: &mut CS) -> Result<(), SynthesisError> {
        let amount = cs.alloc(|| "amount", || self.amount.ok_or(SynthesisError::AssignmentMissing))?;
        cs.enforce(|| "amount > 0", |lc| lc + amount, |lc| lc + CS::one(), |lc| lc);
        Ok(())
    }
}
impl Blockchain {
    // 创建隐私交易
    pub fn create_privacy_transaction(&self, sender: &str, receiver: &str, amount: u64) -> Proof<Bn256> {
        let circuit = PrivacyCircuit {
            amount: Some(amount),
        };
        let params = // 生成 ZKP 参数
        let proof = // 生成 ZKP 证明
        proof
    }
    // 验证隐私交易
    pub fn verify_privacy_transaction(&self, proof: Proof<Bn256>, vk: VerifyingKey<Bn256>) -> bool {
        // 验证 ZKP 证明
        true
    }
}
```

### 可完善的点

- 性能优化：优化 ZKP 的生成和验证性能。
- 范围证明：支持金额的范围证明。
- 批量验证：支持批量验证隐私交易。

## 代码提示

```
RUST
// 扩展范围证明
pub struct RangeProof {
    // 实现范围证明逻辑
}

impl RangeProof {
    pub fn new(amount: u64) -> Self {
        // 初始化范围证明
    }

    pub fn verify(&self) -> bool {
        // 验证范围证明
    }
}

// 批量验证
pub fn batch_verify(transactions: Vec<PrivacyTransaction>) -> bool {
    transactions.iter().all(|tx| tx.verify())
}

// 增强安全性
pub fn secure_execute(&mut self, method: &str, args: Vec<String>) -> Result<String, String> {
    if method == "withdraw" {
        // 防止重入攻击
    }
    self.execute(method, args)
}
```

### 3. CLI 模块 (cli.rs)

B3 扩展了 CLI 模块，支持智能合约和隐私交易的操作。

#### 当前状态

- **功能：**支持部署合约、创建隐私交易等操作。
- **问题：**
  - 功能较为基础，缺乏高级操作支持。
  - 用户体验较差，操作流程不够简化。
  - 缺乏图形化界面 UI 操作

#### 可完善的点

- **功能扩展：**支持更多高级操作，如合约调试、隐私交易查询等。

- **用户体验：**简化操作流程，提供更友好的交互界面。

## 剩余可扩展的部分举例

### 1. P2P 网络模块 (`node.rs`)

- **当前状态：**仅支持基本的节点同步，缺乏节点发现和消息协议。
- **可扩展的点：**
  - **节点发现：**实现 Kademlia DHT 节点发现协议。
  - **消息协议：**定义标准化的网络消息协议。
  - **分片支持：**支持区块链分片技术。

## 代码提示

```
RUST
// 扩展节点发现
pub struct NodeDiscovery {
    // 实现节点发现逻辑
}

impl NodeDiscovery {
    pub fn new() -> Self {
        // 初始化节点发现
    }

    pub fn discover_peers(&self) -> Vec<SocketAddr> {
        // 发现对等节点
    }
}
```

### 2. 性能优化

- **当前状态：**存储和网络性能较为基础，缺乏优化。
- **可扩展的点：**
  - **存储优化：**使用更高效的存储结构，如 Merkle Patricia Trie。
  - **网络优化：**优化 P2P 网络的通信效率，降低延迟。

## 代码提示

```
RUST
// 优化存储
pub fn optimize_storage(&self) {
    // 使用更高效的存储结构
}
```

---

### 3. 安全性增强

- **当前状态：**安全性较为基础，缺乏高级防护机制。
- **可拓展的点：**
  - **防重入攻击：**增强智能合约的安全性。
  - **防双花攻击：**增强交易系统的安全性。

#### 代码提示

```
RUST
// 防重入攻击 pub fn prevent_reentrancy(&mut self) {
    // 实现防重入逻辑
}
```

1. **共识机制：**实现 PoS（权益证明）共识机制。
2. **智能合约：**支持简单的智能合约。
3. **数据隐私：**使用零知识证明（ZKP）技术实现隐私交易。

以下是具体的实施方案和代码。

---

### 4. 修改原有共识机制：例如 PoS（权益证明）

#### 举例说明

- 替换当前的 PoW（工作量证明）机制，实现 PoS 共识。
- 根据持币量选择矿工。

#### 实现步骤

##### 4.1 修改 Blockchain 结构体

在 Blockchain 中添加持币量信息。

```
RUST
use std::collections::HashMap;
#[derive(Debug)]
pub struct Blockchain {
    pub chain: Vec<Block>,
    pub balances: HashMap<String, u64>, // 存储地址余额
}
```

```
pub staking_pool: HashMap<String, u64>, // 存储质押的代币数量
}
```

## 4.2 实现 PoS 挖矿逻辑

根据持币量选择矿工。

```
RUST
impl Blockchain {
    // 选择矿工
    pub fn select_miner(&self) -> Option<String> {
        let total_stake: u64 = self.staking_pool.values().sum();
        if total_stake == 0 {
            return None;
        }
        let mut rng = rand::thread_rng();
        let random_value: u64 = rng.gen_range(0..total_stake);
        let mut cumulative_stake = 0;
        for (address, stake) in &self.staking_pool {
            cumulative_stake += stake;
            if cumulative_stake > random_value {
                return Some(address.clone());
            }
        }
        None
    }

    // 质押代币
    pub fn stake(&mut self, address: &str, amount: u64) {
        let balance = self.get_balance(address);
        if balance >= amount {
            self.set_balance(address, balance - amount);
            *self.staking_pool.entry(address.to_string()).or_insert(0)
+= amount;
        }
    }

    // 取消质押
    pub fn unstake(&mut self, address: &str, amount: u64) {
        if let Some(stake) = self.staking_pool.get_mut(address) {
            if *stake >= amount {
                *stake -= amount;
                let balance = self.get_balance(address);
                self.set_balance(address, balance + amount);
            }
        }
    }
}
```



```
}
```

### 4.3 修改挖矿逻辑

使用 PoS 选择矿工并发放奖励。

```
RUST
impl Blockchain {
    pub fn add_block(&mut self, transactions: Vec<Transaction>) {
        // 选择矿工
        if let Some(miner) = self.select_miner() {
            // 发放挖矿奖励
            let mining_reward = 50; // 挖矿奖励金额
            let reward_transaction = Transaction {
                sender: "0".to_string(), // 系统地址
                receiver: miner.clone(),
                amount: mining_reward,
                signature: "mining_reward".to_string(), // 挖矿奖励不需要
                // 签名
            };
            self.update_balances(&reward_transaction);
            // 创建新区块
            let latest_block = self.get_latest_block();
            let mut new_block = Block::new(
                latest_block.index + 1,
                Utc::now().timestamp(),
                transactions,
                latest_block.hash.clone(),
            );
            new_block.mine_block(self.difficulty);
            self.chain.push(new_block);
        } else {
            println!("No miner selected: staking pool is empty.");
        }
    }
}
```

---

## 5. 代币功能的核心需求

发币的核心是为区块链添加一种原生代币（Native Token），并实现代币的发行、转账和余额管理功能。以下是实现发币功能的一种可行的举例：

### 5.1 代币发行

在创世区块中初始化代币分配。例如，可以给某个地址分配初始代币。

src/blockchain.rs

```
RUST
impl Blockchain {
    pub fn new(difficulty: usize) -> Self {
        let mut blockchain = Blockchain {
            chain: Vec::new(),
            difficulty,
        };
        // 创建创世区块并分配初始代币
        let genesis_transaction = Transaction {
            sender: "0".to_string(), // 系统地址
            receiver: "miner_address".to_string(), // 初始代币分配给矿工
            amount: 1_000_000, // 初始代币总量
            signature: "genesis".to_string(), // 创世交易不需要签名
        };
        let genesis_block = Block::new(0, Utc::now().timestamp(), vec![
genesis_transaction], "0".to_string());
        blockchain.chain.push(genesis_block);
        blockchain
    }
}
```

---

### 5.2 代币转账

在交易执行时，检查发送者的余额并更新余额。

src/blockchain.rs

```
RUST
```

---

```

impl Blockchain {
    pub fn add_block(&mut self, transactions: Vec<Transaction>) {
        // 检查交易的有效性
        for tx in &transactions {
            if !self.is_transaction_valid(&tx) {
                panic!("Invalid transaction: {:?}", tx);
            }
        }
        // 更新余额
        for tx in &transactions {
            self.update_balances(&tx);
        }
        // 创建新区块
        let latest_block = self.get_latest_block();
        let mut new_block = Block::new(
            latest_block.index + 1,
            Utc::now().timestamp(),
            transactions,
            latest_block.hash.clone(),
        );
        new_block.mine_block(self.difficulty);
        self.chain.push(new_block);
    }
    // 检查交易的有效性
    fn is_transaction_valid(&self, tx: &Transaction) -> bool {
        let sender_balance = self.get_balance(&tx.sender);
        sender_balance >= tx.amount
    }
    // 更新余额
    fn update_balances(&mut self, tx: &Transaction) {
        let sender_balance = self.get_balance(&tx.sender);
        let receiver_balance = self.get_balance(&tx.receiver);
        self.set_balance(&tx.sender, sender_balance - tx.amount);
        self.set_balance(&tx.receiver, receiver_balance + tx.amount);
    }
    // 获取地址的余额
    fn get_balance(&self, address: &str) -> u64 {
        // 这里可以使用一个 HashMap 来存储余额
        // 例如: self.balances.get(address).unwrap_or(&0)
        0 // 需要实现具体的余额管理逻辑
    }
    // 设置地址的余额
    fn set_balance(&mut self, address: &str, balance: u64) {
        // 这里可以使用一个 HashMap 来存储余额
    }
}

```

```
        // 例如: self.balances.insert(address.to_string(), balance);
    }
}
```

---

### 5.3 挖矿奖励

在挖出新区块时，给矿工地址发放代币奖励。

src/blockchain.rs

```
RUST
impl Blockchain {
    pub fn add_block(&mut self, transactions: Vec<Transaction>, miner_address: &str) {
        // 检查交易的有效性
        for tx in &transactions {
            if !self.is_transaction_valid(&tx) {
                panic!("Invalid transaction: {:?}" , tx);
            }
        }
        // 更新余额
        for tx in &transactions {
            self.update_balances(&tx);
        }
        // 发放挖矿奖励
        let mining_reward = 50; // 挖矿奖励金额
        let reward_transaction = Transaction {
            sender: "0".to_string(), // 系统地址
            receiver: miner_address.to_string(),
            amount: mining_reward,
            signature: "mining_reward".to_string(), // 挖矿奖励不需要签名
        };
        self.update_balances(&reward_transaction);
        // 创建新区块
        let latest_block = self.get_latest_block();
        let mut new_block = Block::new(
            latest_block.index + 1,
            Utc::now().timestamp(),
            transactions,
            latest_block.hash.clone(),
        );
        new_block.mine_block(self.difficulty);
        self.chain.push(new_block);
    }
}
```

```
}  
}
```

## 5.4 余额管理

使用 `HashMap` 来存储每个地址的余额。

`src/blockchain.rs`

```
RUST  
use std::collections::HashMap;  
#[derive(Debug)]  
pub struct Blockchain {  
    pub chain: Vec<Block>,  
    pub difficulty: usize,  
    pub balances: HashMap<String, u64>, // 存储地址余额  
}  
impl Blockchain {  
    pub fn new(difficulty: usize) -> Self {  
        let mut blockchain = Blockchain {  
            chain: Vec::new(),  
            difficulty,  
            balances: HashMap::new(),  
        };  
        // 创建创世区块并分配初始代币  
        let genesis_transaction = Transaction {  
            sender: "0".to_string(), // 系统地址  
            receiver: "miner_address".to_string(), // 初始代币分配给矿工  
            amount: 1_000_000, // 初始代币总量  
            signature: "genesis".to_string(), // 创世交易不需要签名  
        };  
        blockchain.balances.insert("miner_address".to_string(), 1_000_000);  
        let genesis_block = Block::new(0, Utc::now().timestamp(), vec![  
genesis_transaction], "0".to_string());  
        blockchain.chain.push(genesis_block);  
        blockchain  
    }  
    // 获取地址的余额  
    fn get_balance(&self, address: &str) -> u64 {  
        *self.balances.get(address).unwrap_or(&0)  
    }  
}
```

```
// 设置地址的余额
fn set_balance(&mut self, address: &str, balance: u64) {
    self.balances.insert(address.to_string(), balance);
}
}
```

---

## 5.5. 运行示例

### 创建钱包

```
BASH
cargo run -- create-wallet
```

输出：

```
PLAINTEXT
New wallet created!
Wallet address: 8a7c3f...
```

### 发起交易

```
BASH
cargo run -- add-transaction --sender 8a7c3f... --receiver bob --amount 100
```

输出：

```
PLAINTEXT
Transaction created: Transaction { ... }
```

### 挖矿确认

```
BASH
cargo run -- mine-block --miner miner
```

输出：

```
PLAINTEXT
New block mined by miner: miner
Latest block: Block { ... }
```

### 验证链

```
BASH
cargo run -- validate-chain
```

输出：

PLAINTEXT

Blockchain validity: true

## 总结

B3 在 B1 的基础上新增了智能合约、隐私交易和 CLI 扩展模块，但这些模块目前仍处于半成品状态。通过引入 Gas 机制、范围证明、节点发现等功能，可以显著提升 B3 的功能性和性能。未来可以从智能合约、隐私交易、P2P 网络和性能优化等角度进一步拓展和完善 B3。