基于Rust的简易区块链项目技术方案

架构设计

采用分库设计，以提高系统的可维护性、可扩展性和复用性。以目前的代码量以及功能来说，分为核心库、工具库、以及主程序库。 后续代码量增加，功能复杂后可以采用分库分层的方式更加细化代码。

- 核心库： 负责定义区块、区块链。以及区块的生成、区块链的初始化，区块链代码的增加以及区块链的持久化。工作量证明的代码也存放于核心库中。
- 工具库：hash算法的实现，序列化反序列代码的实现都是存放在工具库中。
- 主程序库: 调用核心库代码，初始化创建区块链，增加交易等。后续可能实现命令行程序进行执行。

代码解读

- 区块定义

```
#[derive(Serialize, Deserialize, Debug)]
pub struct BlockHeader {
    /// 时间戳
    pub time: i64,
    /// 对整个交易数据求hash
    pub tx_hash: String,
    /// 前一条链的hash
    pub pre_hash: String,
}
#[derive(Debug, Serialize, Deserialize)]
pub struct Block {
    pub header: BlockHeader,
    /// 对整个头求hash
    pub hash: String,
    /// 存储的数据
    pub data: String,

    pub nonce: u64,
}
```

- 创建区块

```
impl Block {
    fn set_hash(&mut self) {
        // set_hash的时候进行工作量证明
        let pow = ProofOfWork::new(&self);
        //hash 存的是头的hash
        let (nonce, hash) = pow.run();
        self.hash = hash;
        self.nonce = nonce;
    }

    pub fn new(data: String, pre_hash: String) -> Block {
        // 对数据取hash,按理tx_hash是按照特定方式实现的
        let transactions = coder::serialize_to_bincode(&data);
        // 是整个交易的hash
        let tx_hash = coder::get_hash(&transactions);
```

```
        let mut block = Block {
            header: BlockHeader {
                time: Utc::now().timestamp(),
                tx_hash,
                // 存上一个的hash
                pre_hash,
            },
            // 存的是整个头部的hash
            hash: String::default(),
            data,
            nonce: 0,
        };
        block.set_hash();
        block
    }
}
```

- 工作量证明代码，用于创建区块时计算hash

```
#[derive(Debug)]
pub struct ProofOfWork<'a> {
    pub block: &'a Block,
    pub target: BigUint,
}
```

- 计算工作量代码

```
/// 创建一个新的工作量证明结构体
pub fn new(block: &'a Block) -> Self {
        let mut target = 1.to_biguint().unwrap();
        // 计算左移的位数
        let shift_amount = 256 - TARGET_BITS;
        // 向左移动
        target <<= shift_amount;
        Self { block, target }
    }
    /// 该函数的功能就是将输入的数字与block头进行组合
    fn prepare_data(&self, nonce: u64) -> Vec<u8> {
        let mut data = coder::serialize_to_bincode(&self.block.header);
        // 然后将target_bits加进入
        data.append(&mut coder::serialize_to_bincode(&TARGET_BITS));
        // 将数组加进入
        data.append(&mut coder::serialize_to_bincode(&nonce));
        data
    }

    /// 实现计算算法，目的是找到一个数字，实现hash以后前几位是target_bits位是0
    pub fn run(&self) -> (u64, String) {
        let mut hash_int: BigUint;
        let mut hash = String::default();
        let mut nonce = 0;
        while nonce < MAX {
            let data = self.prepare_data(nonce);
```

```
            hash = coder::get_hash(&data[..]);
            hash_int = BigUint::parse_bytes(hash.as_bytes(), 16).unwrap();
            if hash_int < self.target {
                break;
            } else {
                nonce += 1;
            }
        }
        (nonce, hash)
    }
```

- 区块链的定义以及新增

```
/// 区块链定义
pub struct BlockChain {
    /// 是DB中存储的最后一个区块
    pub tip: String,
    /// 遍历的时候用到
    current_hash: String,
    /// 数据库链接
    db: DB,
}
/// 区块链新增
pub fn new() -> Self {
        // 第一步是检查 open_default该方法是如果没有数据库就创建
        let db = DB::open_default(DB_DIR).unwrap();
        // 查看该数据库中是否存储了相关数据
        let tip = match db.get(BLOCKCHAIN_TARGET).unwrap() {
            None => {
                //如果没有1，那就说明这个里面不存在数据，那我就创建
                let genesis = Self::new_genesis_block();
                // 将hash存下来
                let _ = db.put(genesis.hash.clone(),
coder::serialize_to_bincode(&genesis));
                // 将l存下来
                let _ = db.put(BLOCKCHAIN_TARGET, genesis.hash.clone());
                genesis.hash
            }
            Some(value) => {
                // 有的话就直接取出来，
                String::from_utf8(value).unwrap()
            }
        };
        Self {
            tip: tip.clone(),
            db,
            current_hash: tip,
        }
    }
```

- 创建创世区块

```
/// 创建创世区块
 fn new_genesis_block() -> Block {
    Block::new("This is genesis block".to_string(),String::default())
}
```

- 向区块链中增加区块

```
pub fn add(&mut self, data: String) {
        let new_block = Block::new(data, self.tip.clone());
        // 更新数据
        self.db
            .put(
                new_block.hash.clone(),
                coder::serialize_to_bincode(&new_block),
            )
            .unwrap();
        self.db
            .put(BLOCKCHAIN_TARGET, new_block.hash.clone())
            .unwrap();
        // 然后处理数据
        self.tip = new_block.hash.clone();
        // 指向最新的
        self.current_hash = new_block.hash.clone();
    }
```

- 遍历区块链

```
impl Iterator for BlockChain {
    type Item = Block;

    fn next(&mut self) -> Option<Self::Item> {
        // 实现的迭代器
        match self.db.get(self.current_hash.as_bytes()) {
            Ok(raw_value) => match raw_value {
                None => None,
                Some(value) => {
                    // 反序列化
                    let block: Block = coder::bincode_deserialize(&value[..]);
                    self.current_hash = block.header.pre_hash.clone();
                    Some(block)
                }
            },
            Err(_) => None,
        }
    }
}
```