技术方案

1. 技术选型

- 开发语言：Rust（^1.85.0）
- 加密算法：hex、crypto-hash
- 测试框架：cargo-test
- 测试工具：Postman/RustRover
- 服务端框架：actix-web

2. 核心数据结构

2.1 Block 结构体

定义在 src/block.rs 中，表示区块链中的一个区块。包含以下字段：

- index：区块索引
- timestamp：时间戳
- hash：区块哈希
- prev_block_hash：前一个区块的哈希
- nonce：随机数
- transactions：交易列表
- difficulty：挖矿难度

```rust
pub struct Block {
    pub index: u32,
    pub timestamp: u128,
    pub hash: Hash,
    pub prev_block_hash: Hash,
    pub nonce: u64,
    pub transactions: Vec<Transaction>,
    pub difficulty: u128,
}
```

2.2 Blockchain 结构体

定义在 src/blockchain.rs 中，表示区块链。包含以下字段：

- blocks：区块列表
- unspent_outputs：未花费输出集合

```
// 定义区块链结构体
pub struct Blockchain {
    pub blocks: Vec<Block>,
    unspent_outputs: HashSet<Hash>,
}
```

2.3 Transaction 结构体

定义在 src/transaction.rs 中，表示交易。包含以下字段：

- inputs：输入列表
- outputs：输出列表

```
// 定义交易结构体
pub struct Transaction {
    pub inputs: Vec<Output>,
    pub outputs: Vec<Output>,
}
```

2.4 Hashable 特性

定义在 src/hashable.rs 中，表示可哈希的对象。包含以下方法：

- bytes：返回对象的字节表示
- hash：返回对象的哈希值

```
pub trait Hashable {
    fn bytes(&self) -> Vec<u8>;

    fn hash(&self) -> Hash {
        crypto_hash::digest(crypto_hash::Algorithm::SHA256, &self.bytes
())
    }
}
```

2.5 工具函数

定义在 src/lib.rs 中，包括：

- now：获取当前时间戳
- u32_bytes：将 u32 转换为字节数组
- u64_bytes：将 u64 转换为字节数组
- u128_bytes：将 u128 转换为字节数组
- difficulty_bytes_as_u128：将字节数组转换为 u128

```
pub fn now() -> u128 {
    let duration = SystemTime::now().duration_since(UNIX_EPOCH).unwrap
();

    // 返回时间戳
    duration.as_secs() as u128 * 1000 + duration.subsec_millis() as u12
8
}

// 获取 u32 类型的字节数组
pub fn u32_bytes(u: &u32) -> [u8; 4] {
    [
        (u >> 8 * 0x0) as u8,
        (u >> 8 * 0x1) as u8,
        (u >> 8 * 0x2) as u8,
        (u >> 8 * 0x3) as u8,
    ]
}

// 获取 u64 类型的字节数组
pub fn u64_bytes(u: &u64) -> [u8; 8] {
    [
        (u >> 8 * 0x0) as u8,
        (u >> 8 * 0x1) as u8,
        (u >> 8 * 0x2) as u8,
        (u >> 8 * 0x3) as u8,
        (u >> 8 * 0x4) as u8,
        (u >> 8 * 0x5) as u8,
        (u >> 8 * 0x6) as u8,
        (u >> 8 * 0x7) as u8,
    ]
}

// 获取 u128 类型的字节数组
pub fn u128_bytes(u: &u128) -> [u8; 16] {
    [
        (u >> 8 * 0x0) as u8,
        (u >> 8 * 0x1) as u8,
```

```
        (u >> 8 * 0x2) as u8,
        (u >> 8 * 0x3) as u8,
        (u >> 8 * 0x4) as u8,
        (u >> 8 * 0x5) as u8,
        (u >> 8 * 0x6) as u8,
        (u >> 8 * 0x7) as u8,
        (u >> 8 * 0x8) as u8,
        (u >> 8 * 0x9) as u8,
        (u >> 8 * 0xa) as u8,
        (u >> 8 * 0xb) as u8,
        (u >> 8 * 0xc) as u8,
        (u >> 8 * 0xd) as u8,
        (u >> 8 * 0xe) as u8,
        (u >> 8 * 0xf) as u8,
    ]
}

// 获取困难度字节数组
pub fn difficulty_bytes_as_u128(v: &Vec<u8>) -> u128 {
    assert!(
        v.len() >= 16,
        "The input vector must have at least 16 bytes"
    );

    let mut result = 0u128;
    for (i, byte) in v.iter().rev().take(16).enumerate() {
        result |= (*byte as u128) << (i * 8);
    }
    result
}
```

3. 关键算法

3.1 更新区块链

```
pub fn update_with_block(&mut self, block: Block) -> Result<(), BlockVa
lidationErr> {
    let block_num = self.blocks.len();

    // 检查区块是否有效
    if block.index != block_num as u32 {
        return Err(BlockValidationErr::MismatchedIndex);
    } else if !block::check_difficulty(&block.hash(), block.difficulty)
 {
        return Err(BlockValidationErr::InvalidHash);
    } else if block_num != 0 {
        // 非 Genesis 区块
        let prev_block = &self.blocks[block_num - 1];
        if block.timestamp <= prev_block.timestamp {
```

```
                return Err(BlockValidationErr::AchronologicalTimestamp);
            } else if block.prev_block_hash != prev_block.hash {
                return Err(BlockValidationErr::MismatchedPreviousHash);
            }
        } else {
            // Genesis 区块
            if block.prev_block_hash != vec![0; 32] {
                return Err(BlockValidationErr::InvalidGenesisBlockFormat);
            }
        }

        // 检查交易是否有效
        if let Some((coinbase, transactions)) = block.transactions.split_fi
rst() {
            // 检查 Coinbase 交易
            if !coinbase.is_coinbase() {
                return Err(BlockValidationErr::InvalidCoinbaseTransaction);
            }

            // 检查双花问题
            let mut block_spent: HashSet<Hash> = HashSet::new();
            let mut block_created: HashSet<Hash> = HashSet::new();
            let mut total_fee = 0;

            // 遍历区块中的交易
            for transaction in transactions {
                let input_hashes = transaction.input_hashes();

                // 检查输入是否有效且未被重复花费
                if !(&input_hashes - &self.unspent_outputs).is_empty()
                    || !(&input_hashes & &block_spent).is_empty()
                {
                    return Err(BlockValidationErr::InvalidInput);
                }

                // 计算输入和输出金额
                let input_value = transaction.input_value();
                let output_value = transaction.output_value();

                // 输出金额不可超过输入金额
                if output_value > input_value {
                    return Err(BlockValidationErr::InsufficientInputValue);
                }

                // 累加手续费
                let fee = input_value - output_value;
                total_fee += fee;
```

```
            // 记录已花费和新生成的 UTXO
            block_spent.extend(input_hashes);
            block_created.extend(transaction.output_hashes());
        }

        // Coinbase 交易必须覆盖手续费
        if coinbase.output_value() < total_fee {
            return Err(BlockValidationErr::InvalidCoinbaseTransaction);
        } else {
            block_created.extend(coinbase.output_hashes());
        }

        // 更新 UTXO 集合
        self.unspent_outputs
            .retain(|output| !block_spent.contains(output));
        self.unspent_outputs.extend(block_created);
    }

    self.blocks.push(block);

    Ok(())
}
```

3.2 挖矿算法

```
pub fn mine(&mut self) -> std::io::Result<()> {
        for nonce_attempt in 0..u64::MAX {
            self.nonce = nonce_attempt;
            let hash = self.hash();
            if check_difficulty(&hash, self.difficulty) {
                self.hash = hash;
                return Ok(());
            }
        }
        Err(std::io::Error::new(ErrorKind::InvalidData, "Couldn't find
valid hash"))
    }
```

3.3 创建创世区块链

```
fn create_genesis_blockchain(difficulty: u128) -> Blockchain {
    let mut genesis_block = Block::new(
        0,
        now(),
        vec![0; 32],
        vec![Transaction {
            inputs: vec![],
            outputs: vec![
                transaction::Output {
```

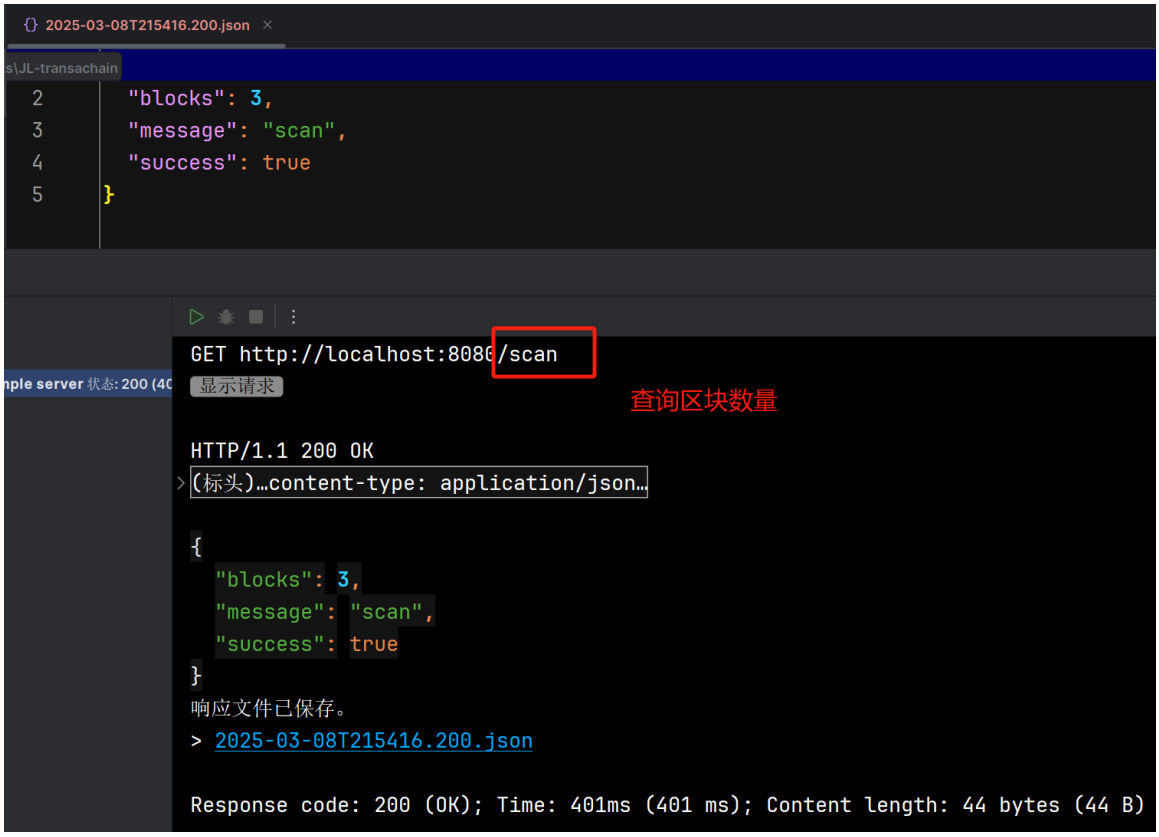
```
                    receiver: "Alice".to_owned(),
                    value: 50,
                },
                transaction::Output {
                    receiver: "Bob".to_owned(),
                    value: 7,
                },
            ],
        }],
        difficulty,
    );
    genesis_block.mine();

    let mut blockchain = Blockchain::new();
    blockchain.update_with_block(genesis_block).unwrap();
    blockchain
}
```

4. web 接口调试

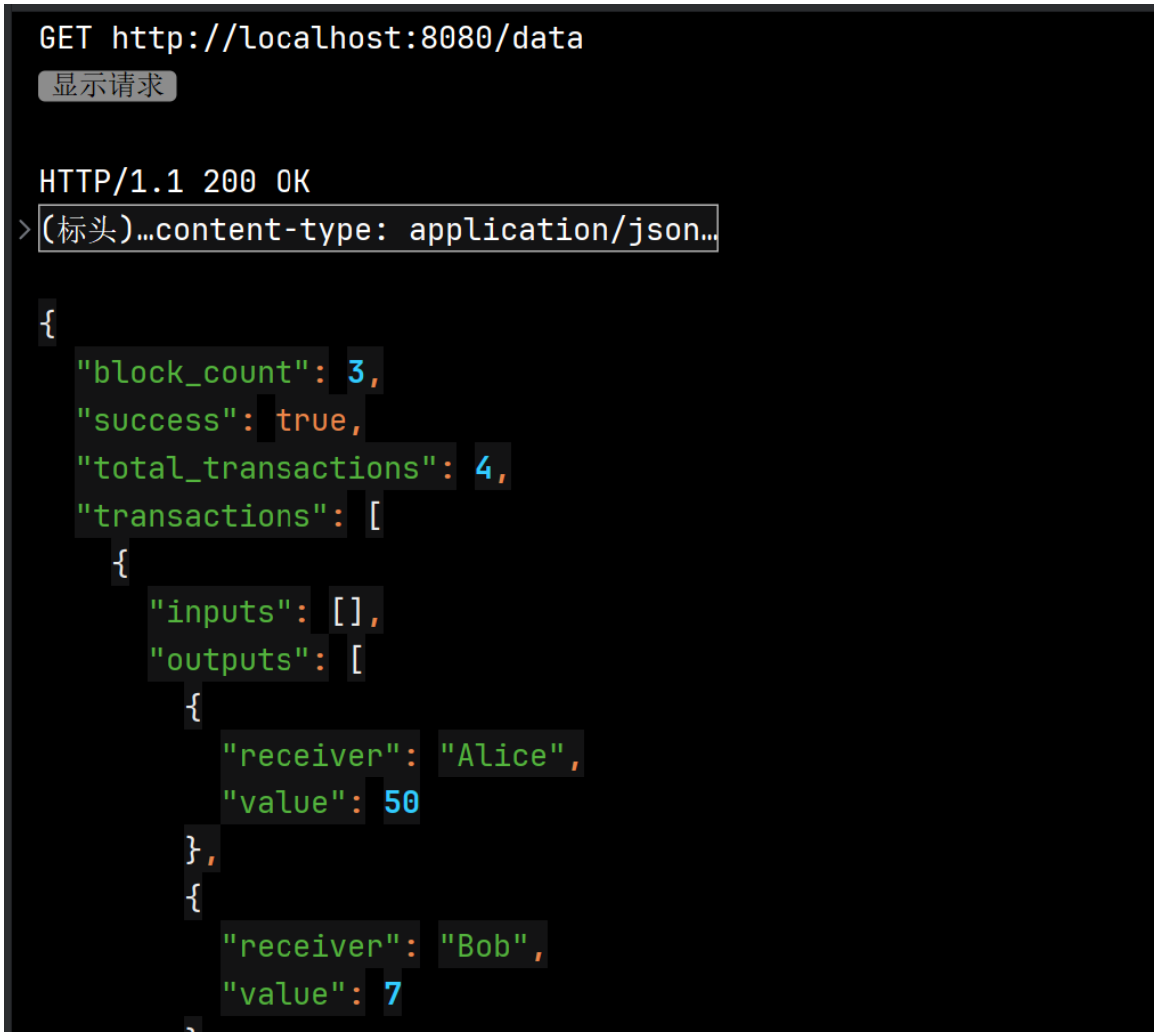
4.1 scan 接口

访问 web 提供的接口/scan



4.2 data 接口

访问 web 提供的接口/data



4.3 mine 接口

访问 web 提供的接口`/mine`