技术方案

技术栈

1. **编程语言**
   - **Rust**：强调内存安全与高性能，适合实现底层区块链逻辑。
2. **核心依赖库**
   - `serde`：用于结构体序列化与反序列化。
   - `chrono`：处理区块时间戳（UTC标准时间）。
   - `crypto`：提供SHA3-256哈希算法。
   - `bincode`：二进制序列化工具，高效处理数据结构。

关键Rust语法与功能

1. **结构体与派生宏**

```rust
#[derive(Serialize, Deserialize, Debug, PartialEq, Eq)]
pub struct BlockHeader { /* ... */ }
```

   - 通过`serde`宏实现自动序列化，支持跨模块数据传输。
2. **所有权与可变性**
   - 在挖矿过程中使用`&mut self`修改区块头Nonce值：

```rust
pub fn mine_block(&mut self, difficulty: usize) { /* ... */ }
```

3. **错误处理**
   - 使用`unwrap`简化序列化/反序列化错误处理（注：生产环境建议改用`Result`）。
4. **模块化组织**
   - 通过`pub mod`公开模块（`block`,`blockchain`,`coder`），实现代码分层。

核心算法实现

1. **工作量证明（PoW）**

```rust
pub fn mine_block(&mut self, difficulty: usize) {
    let prefix = "0".repeat(difficulty);
    while !self.hash.starts_with(&prefix) {
        self.header.nonce += 1;
        self.set_hash(); // 重新计算哈希
    }
}
```

   - 通过循环调整Nonce，直到哈希满足前导零条件。

2. **哈希计算**
    - 使用 `crypto::sha3::Sha3` 生成SHA3-256哈希值，确保抗碰撞性。

性能优化点

1. **序列化效率**
    - 采用 `bincode` 替代JSON，减少序列化后的数据体积。
2. **难度控制**
    - 通过常量 `DIFFICULTY` 集中管理挖矿复杂度（示例值为4）：

```
impl BlockChain {
    const DIFFICULTY: usize = 4;
}
```

后续改进建议

1. **动态难度调整**
    - 根据全网算力自动调整 `DIFFICULTY` 值。
2. **并发挖矿**
    - 使用Rust的 `tokio` 或 `rayon` 库实现多线程Nonce搜索。
3. **区块验证**
    - 添加区块验证功能防止交易篡改