接口文档

1. 区块链底层组 (Blockchain Core)

成员 1: 区块链核心开发 (数据层)

核心职责: 定义数据结构，维护链的完整性。

阶段	接口名	输入	输出	处理逻辑
数据定义	`Transaction`, `Block`, `Account` (类定义)	无	`Transaction`, `Block`, `Account` 类	定义 Pydantic 模型或 Python 类。
哈希计算	`calculate_hash`, `get_merkle_root`	任意字符串/字节流	哈希值 (String)	实现 SHA256 哈希算法；实现 Merkle Tree 根哈希计算。
链维护	`Blockchain.add_block`	`Block` 对象 (来自成员 2 的挖矿结果)	更新后的区块链列表	验证区块哈希链接 (`prev_hash`)；将合法区块追加到 `self.chain` 列表。
公钥管理	`PublicKeyRegistry.get_public_key`	地址 (String)	公钥 (String)	根据账户地址查找对应的公钥，供成员 2 进行签名验证使用。
地址生成	`generate_address`	公钥 (bytes)	地址 (String)	实现从公钥生成账户地址的算法（如 SHA256 + 截取）。

成员 2: 区块链核心开发 (执行层)

核心职责: 交易执行与状态更新。

阶段	接口名	输入	输出	处理逻辑
交易池管理	`Blockchain.add_transaction`	`Transaction` 对象	bool	验证交易签名 (`self._verify_transaction_signature(tx)`)；检查 Nonce 防止重放；加入 `pending_transactions` 。
挖矿/出块	`Blockchain.mine_block`	无显式参数	`Optional[Block]`	从池中取出交易；调用 `state_processor.apply_transaction()` 执行交易；打包生成新 `Block` 。
状态更新	`StateProcessor.apply_transaction`	`Transaction` 对象	bool	解析交易类型；根据不同交易类型调用对应处理函数更新全局状态 (`WorldState`)。

1. 交易池管理接口

接口名称: `Blockchain.add_transaction`

功能描述: 将交易添加到待处理交易池中，等待被打包进区块。

输入参数:

- `tx` (Transaction): 交易对象，包含交易类型、发送者地址、nonce、gas 价格、gas 限制、数据和签名等信息

返回值:

- `bool` : 交易是否成功添加到交易池

处理逻辑:

1. 验证交易签名的有效性，确保交易确实由发送者发起，使用 `self._verify_transaction_signature(tx)` 方法

2. 检查交易的 nonce 值是否与发送者账户的 nonce 匹配，防止重放攻击
3. 验证交易的 gas 限制是否满足最低要求
4. 检查发送者的账户余额是否足够支付交易所需的 gas 费用
5. 若以上验证均通过，则将交易添加到待处理交易池(`pending_transactions`)中

2. 挖矿/出块接口

接口名称: `Blockchain.mine_block`

功能描述: 从交易池中取出待处理交易，执行这些交易并生成新的区块。

输入参数: 无显式参数，内部使用交易池状态

返回值:

- `Optional[Block]`：成功挖矿时返回新生成的区块对象，若没有待处理交易则返回 None

处理逻辑:

1. 检查是否存在待处理交易，若无则直接返回
2. 从交易池中复制所有待处理交易，并清空交易池
3. 为每个交易扣除相应的 gas 费用
4. 调用 `state_processor.apply_transaction()` 执行每笔交易
5. 对于成功执行的交易，增加发送者账户的 nonce 值
6. 对于执行失败的交易，退还已扣除的 gas 费用
7. 使用成功执行的交易创建新区块，并计算区块哈希
8. 将新区块添加到区块链中

3. 状态更新接口

接口名称: `StateProcessor.apply_transaction`

功能描述: 根据交易类型执行相应的状态更新操作。

输入参数:

- `tx`（Transaction): 需要被执行的交易对象

返回值:

- `bool`：交易是否成功应用

处理逻辑:

1. 根据交易对象中的 `tx_type` 字段确定交易类型
2. 若为根因提案交易(`propose_root_cause`)，调用 `_apply_propose_root_cause` 方法创建新的根因提案
3. 若为投票交易(`vote`)，调用 `_apply_vote` 方法处理投票
4. 若为转账交易(`transfer`)，调用 `_apply_transfer` 方法执行转账
5. 根据交易类型调用相应的方法更新世界状态(`WorldState`)
6. 将更新后的状态持久化存储到数据库中

2. 智能合约组 (Smart Contracts)

成员 3: 智能合约开发 (流程控制)

核心职责: 运维 SOP 状态机逻辑。

1. 核心交易方法（Agent 通过交易调用，由 VM 分发）

方法名	对应交易类型（tx_type）	参数	返回值	调用条件（当前状态）	功能描述

`submit_data_collection`	`"submit_data_collection"`	`agent_id: str`
		`data_summary: str`
		`raw_data: Optional[Dict] = None`
			`dict`（包含 `new_state`，`message`）	**Init**	提交故障数据采集结
					推进至数据采集完成

`propose_root_cause`	`"propose_root_cause"`	`agent_id: str` `content: str`	`dict`（包含 `proposal_id`，`new_state`，`message`）	**Data_Collected**	提出根因分析提案， 推进至根因提案阶段

2. 内部协作方法（仅供治理合约调用）

方法名	参数	返回值	调用条件（当前状态）	功能描述	发射事件
`advance_to_consensus_phase`	`proposal_id: str` `passed: bool`	无 （直接更新状态）	**Root_Cause_Proposed**	根据成员 4 的共识结果推进状态 通过 → Consensus → Solution 否决 → 回退 Data_Collected	`ConsensusReached`（pass `SolutionPhaseEntered`（ `ProposalRejected`（否决

3. 查询方法（供前端、API、监控使用，只读）

方法名	参数	返回值	描述
`get_current_state()`	无	`str`（如 "Data_Collected"）	获取当前 SOP 所处阶段
`get_events(limit: int = 50)`	`limit: int`（可选）	`List[Dict]`（最近的事件列表，倒序）	获取事件日志，用于审计和实时展示
`get_current_proposal()`	无	`Optional[Dict]` （当前提案详情，或 None）	获取当前活跃提案（proposer、content 等）
`get_incident_data()`	无	`Dict`（数据采集阶段提交的信息）	获取本次故障的数据采集摘要和原始数据

4. 事件类型（Event）列表（供前端订阅/展示，监听奖惩触发）

事件名	携带字段示例	触发时机
`DataCollected`	`agent_id`，`summary`	数据采集提交成功
`RootCauseProposed`	`proposal_id`，`proposer`，`content`	根因提案提交成功
`ConsensusReached`	`proposal_id`，`passed: bool`	成员 4 调用 advance 后（无论通过或否决）
`SolutionPhaseEntered`	`proposal_id`，`root_cause`	提案通过，进入解决方案阶段
`ProposalRejected`	`proposal_id`，`proposer`	提案被否决

5. 测试辅助方法（仅单元测试使用）

方法名	描述
`reset_for_testing()`	重置所有状态为 Init（仅测试环境使用）

成员 4: 智能合约开发 (经济模型)

核心职责: Token 管理与共识计算。

阶段	接口名	输入	输出	处理逻辑
Token 管理	`TokenContract.transfer`，`TokenContract.stake`，`TokenContract.slash`	`Transaction`（类型: `TRANSFER`，`STAKE`）	更新后的 `balances` 字典	检查余额；执行转账/质押/扣除操作；更新账本。
共识投票	`GovernanceContract.vote`	`Transaction`（类型: `VOTE`）	提案状态（`Passed` / `Rejected`）	记录投票；计算权重（`质押量 * 信誉分`）；判断是否达到阈值。

3. Agent 适配组 (Agent Adapter)

成员 5: Agent 中间件开发

核心职责: 链上交互中间件。

阶段	接口名	输入	输出	处理逻辑
交易封装	`Transaction.sign`（调用）	Agent 的决策（`Action`，`Payload`）	签名后的 `Transaction` 对象	构造 `Transaction` 对象；调用 `sign()` 签名。
上链交互	`ChainClient.send_action`	`Transaction` 对象	交易回执（`Receipt`）	调用成员 2 的 `add_transaction` 接口发送交易；轮询等待回执。

成员 6: Agent 适配与 Prompt 工程

核心职责: Agent 身份与 Prompt 改造。

阶段	接口名	输入	输出	处理逻辑
身份管理	`AgentProfile`（数据结构）	无	`AgentProfile`（包含地址、私钥）	生成/加载钱包私钥；为每个 Agent 实例分配身份。
Prompt 改造	`System Prompt`	原始运维任务描述	结构化的 Agent 输出 (包含 `action_type`，`stake_amount`)	修改 System Prompt，注入“质押”、“经济激励”等约束；解析 LLM 输出为 JSON。

4. 前端交互组 (Frontend & API)

成员 7: 前端开发 (区块链浏览器)

核心职责: 区块链数据可视化。

阶段	接口名	输入	输出	处理逻辑
后端 API 开发	`GET /api/blocks`，`GET /api/transaction/{hash}`	`Blockchain` 对象 (来自成员 1/2)	JSON 数据	封装 FastAPI 接口，暴露区块和交易数据。
前端展示	`BlockExplorer` 组件	API 返回的 JSON 数据	渲染区块列表、交易详情页、Merkle 树验证视图	调用 API 获取数据并渲染。

成员 8: 前端开发 (运维控制台)

核心职责: 业务状态与经济看板。

阶段	接口名	输入	输出	处理逻辑
后端 API 开发	`GET /api/state/sop`，`GET /api/state/agents`	`OpsContract`，`TokenContract` 状态 (来自成员 3/4)	JSON 数据	封装 FastAPI 接口，暴露 SOP 状态和 Agent 资产信息。
前端展示	`SOPDashboard`，`EconomyMonitor` 组件	API 返回的 JSON 数据	渲染 SOP 流程图 (React Flow)、资产图表 (ECharts)	调用 API 获取数据并渲染。