# 骨干链子链管理系统设计方案

# 文档信息

| **项目名称:** | **项目编号:** |
| --- | --- |
| 项目负责人: | 所属部门: |
| 编 制 人: | 编制时间: |
| 审 核 人: | 审核时间: |
| 批 准 人: | 批准时间: |
| 版 本 号: | 流 水 号: |

# 修改记录

| **日期** | **版本** | **修改说明** | **修改者** | | |
| --- | --- | --- | --- | --- | --- |
| | | | | | |
| | | | | | |
| | | | | | |

# 1 系统概述

本系统是骨干链子链管理系统，协助主链管理骨干链链群。通过合约创建链群普通管理员，普通管理员有权申请成为超级管理员，经审核通过后帮助owner一起管理普通管理员。主链普通管理员有权查看所有注册在此合约的骨干链链群信息，而单独骨干链的单位或企业，只能查看自己链在此合约中的信息。同时为防止owner密钥丢失的情况发生，考虑添加owner的继承者以防止意外事件。

# 2 总体架构设计

## 2.1 总体技术架构

针对子链管理系统需要迭代更新的需求，本方案采用可升级的智能合约框架。通过调研，有以下几种方案：

**1.主从合约(Master-Slave contracts)**

部署一个主合约，以及其他合约，其中主合约负责存储所有其他合约的地址，并在需要时返回所需的地址。

优点：简单

缺点：不易进行合约资产转移到新合约

**2.永久存储合约(Eternal Storage contracts)**

逻辑合约和数据合约彼此分开。 数据合约是永久性的，不可升级， 逻辑合约可以根据需要多次升级，并将更改通知给数据合约。

缺点：数据合约不可更改、逻辑合约外部调用数据合约将消耗额外的gas。

### 3. 可升级存储代理合约(UpGradable Storage Proxy Contracts)

代理模式使得所有消息调用都通过代理合约，代理合约会将调用请求重定向到最新部署的合约中。如要升级时，将升级后新合约地址更新到代理合约中即可。


我们可通过使永久存储合约充当逻辑合约的代理，以此防止支付额外的gas。这个代理合约，以及这个逻辑合约，将继承同一存储合约，那么它们的存储会在EVM虚拟机中对齐。这个代理合约将有一个**回退函数**，它将委托调用这个逻辑合约，那么这个逻辑合约就可以在代理存储中进行更改。这个代理合约将是永恒的。这节省了对存储合约多次调用所需的gas，不管数据做了多少的更改，就只需要一次委托调用。

这项技术当中有三个组成部分：

1. **代理合约（Proxy contract）**：它将充当永久存储并负责委托调用逻辑合约；
2. **逻辑合约（Logic contract）**：它负责完成处理所有的数据；
3. **存储结构（Storage structure）**：它包含了存储结构，并会由代理合约和逻辑合约所继承，以便它们的存储指针能够在区块链上保持同步；

**三种代理模式：**

1. 继承存储
2. 永久存储
3. 非结构化存储

这三种模式底层都依赖委托调用 DELEGATECALL 操作码来实现。DELEGATECALL是EVM提供的用于程序集的操作码。它的工作方式与普通调用类似，只是目标地址的代码是在调用合约的上下文中执行的。例如，每当合约 A 将调用代理到另一个合同 B 时，它都会在合约 A 的上下文中执行合约 B 的代码。这意味着将保留 msg.value 和 msg.sender 值，并且每次存储修改都会影响合约 A。

汇编操作码:

```
assembly {
    let ptr := mload(0x40) //它包含了下一个可用的空闲内存指针的值。每次将变量直接保存到内存时，都应通过查询 0x40 位置的值，来确定变量保存在内存的位置。
    calldatacopy(ptr, 0, calldatasize) //calldatasize 获得 msg.data 的大小，使用 calldatacopy 将其复制到 ptr 变量中。

    let result := delegatecall(gas, _impl, ptr, calldatasize, 0, 0)
    //gas 我们传递执行合约所需要燃料
    //_impl 所请求的目标合约地址
    //ptr 请求数据在内存中的起始位置
    //calldatasize 请求数据的大小
    //0 用于表示目标合约的返回值。这是未使用的，因为此时我们尚不知道返回数据的大小，因此无法将其分配给变量。之后我们可以使用 returndata 操作码访问此信息。
    //0 表示目标合约返回值的大小。这是未使用的，因为在调用目标合约之前，我们是无法知道返回值的大小。之后我们可以通过 returndatasize  操作码来获得该值。

    let size := returndatasize //获取返回值的大小
    returndatacopy(ptr, 0, size)  //将返回的数据拷贝到 ptr 变量中

    switch result //switch 语句返回的数据或者抛出异常
    case 0 { revert(ptr, size) }
    default { return(ptr, size) }
 }
```

这三种模式都用来解决同一个难题：**如何确保目标合约不会覆盖代理合约中用于升级的状态变量。**

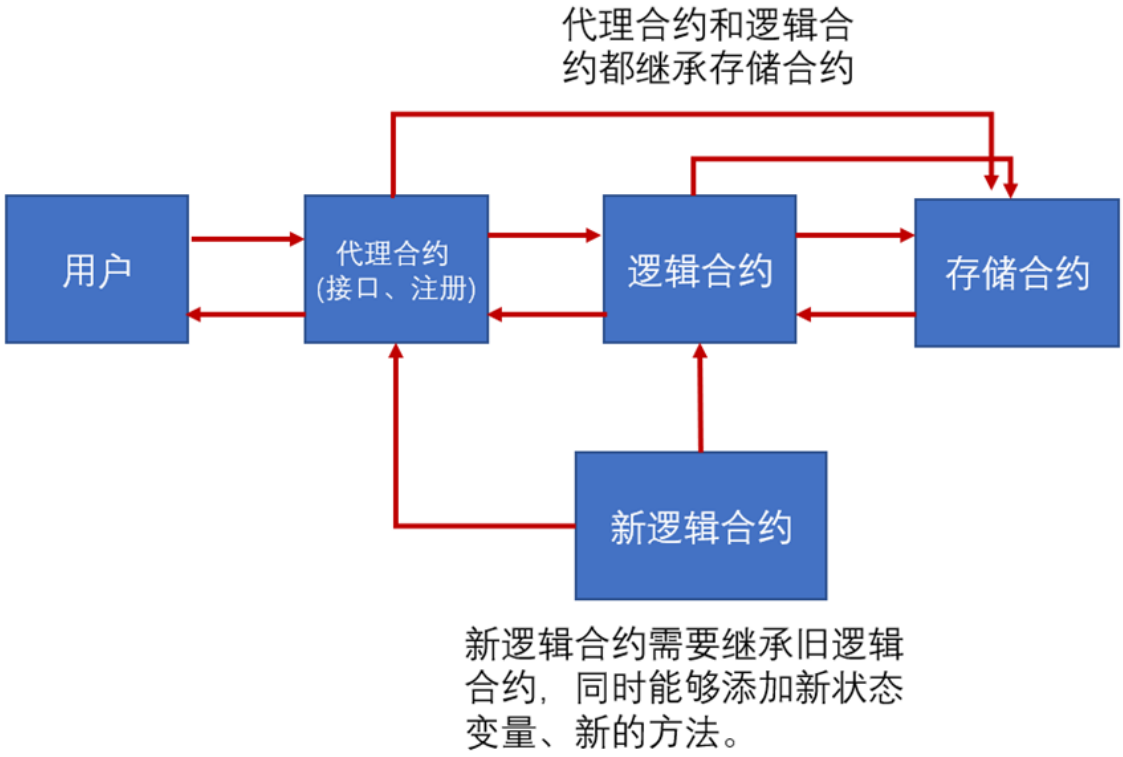
所有代理模式的主要关注点是如何处理存储分配。由于我们将一个合约用于存储，而将另一个合约用于逻辑处理，因此任何一个合约都可能覆盖已使用的存储插槽。这意味着，如果代理合约具有状态变量以跟踪某个存储插槽中的最新逻辑合约地址，而该逻辑合约不知道该变量，则该逻辑合约可能会在同一插槽中存储一些其他数据，从而覆盖代理的关键信息。

### 3.1 继承存储

继承存储方式需要**逻辑合约包含代理合约所需的存储结构。**代理和逻辑合约都继承相同的存储结构，以确保两者都存储必要的代理状态变量。

对于这种方式，使用 `Registry` 合约来跟踪逻辑合同的不同版本。为了升级到新的逻辑合同，开发者需要在注册合约中将新升级的合约进行注册，并要求代理升级到新合约。**新版本的逻辑合约需要继承前一个版本逻辑合约的存储结构。**

**重点：**仍然可以通过 `UpgradeabilityProxy` 合约，来调用新版本目标合约引入的新方法或新变量。*




### 3.2 永久存储

**存储结构是在单独的合约中定义**，代理合约和逻辑合约都继承存储合约。存储合约包含逻辑合约所需的所有状态变量，同时，代理合约也能够识别这些状态变量，因此代理合约在定义升级所需要的状态变量时，不必担心所定义的状态变量会被覆盖。**注意，逻辑合约的后续版本均不应定义任何其他状态变量。逻辑合约的所有版本都必须始终使用最开始定义存储结构。**

**重点：**新版owner可以升级现有合约的方法或引入新的方法，但是不能引入新的状态变量。

### 3.3 非结构化存储

非结构化存储模式类似继承存储模式，但并不需要目标合约继承与升级相关的任何状态变量。此模式使用**代理合约中定义的非结构化存储插槽来保存升级所需的数据。**

在代理合约中，我们定义了一个常量变量，在对它进行 Hash 时，应提供足够随机的存储位置来存储代理合约调用逻辑合约的地址。

```
bytes32 private constant implementationPosition =
keccak256("org.zeppelinos.proxy.implementation");
```

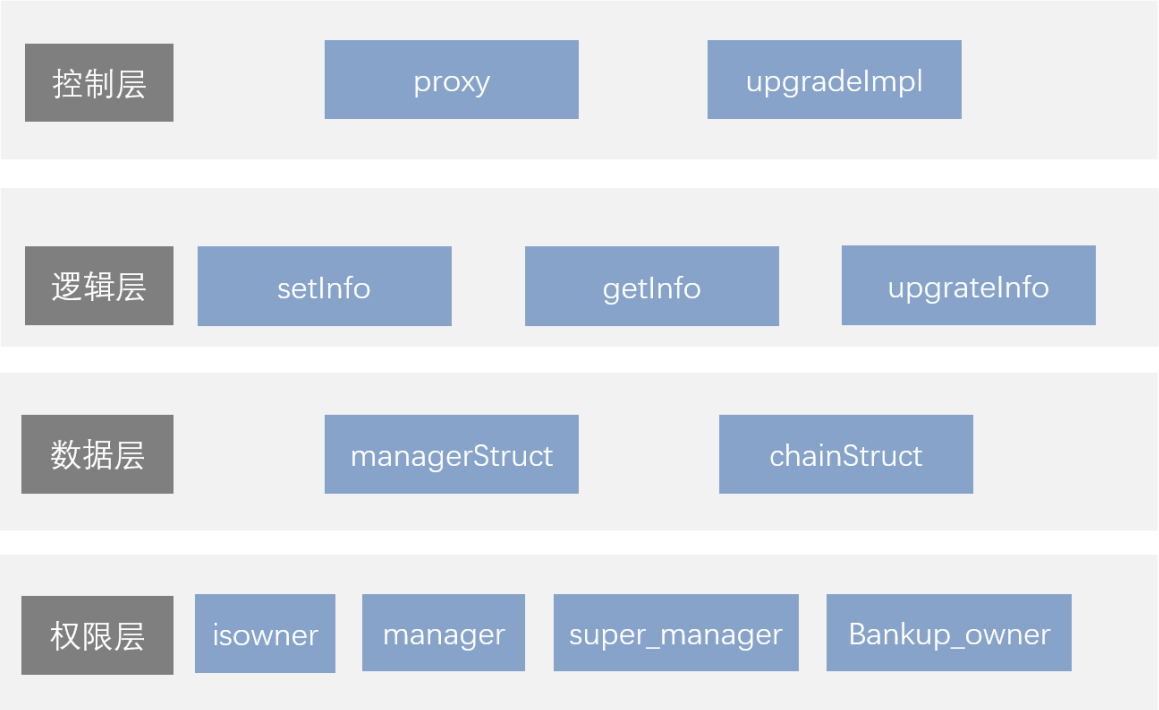
由于常量不会占用存储插槽，因此不必担心 `implementationPosition` 被目标合约意外覆盖。由于 Solidity 状态变量存储的规定，目标合约中定义的其他内容使用此存储插槽冲突的可能性极小。

通过这种模式，**逻辑合约不需要知道代理合约的存储结构**，但是所有**未来的逻辑合约都必须继承其初始版本定义的存储变量**。就像在继承存储模式中一样，将来升级的目标合约可以升级现有功能以及**引入新功能和新存储变量**。

非结构化存储可能是当前最大的可升级性方法，它使我们能够利用存储中状态变量的布局。该技术的工作原理是将可升级性所需的数据保存在存储中的固定位置，以防止被新数据覆盖。可以使用SLOAD和SSTORE操作码进行汇编。由于存储插槽只是从0x0开始递增，因此我们使用很高的存储插槽来防止覆盖我们可以通过对常量变量进行散列来生成存储槽。 由于恒定状态变量不会占用存储空间，因此我们不必担心它会被覆盖。

**重点**：目标合约与代理合约耦合性最低。

### 2.2 总体功能架构




- **控制层**：合约的代理，控制层将请求通过代理转发给逻辑层，逻辑层按照业务逻辑处理后通过数据层进行数据上链；
- **逻辑层**：具体实现骨干链新的的注册、查询、更新；
- **数据层**：管理员信息、骨干链信息、子链信息的具体数据结构；
- **权限层**：合约的管理控制，针对逻辑层的注册、查询、更新，赋予权限管控；

## 3 详细设计

### 3.1 代理合约

```
contract Proxy is StorageStructure {

    //确保只有所有者可以运行这个函数
    modifier onlyOwner() {
        require (msg.sender == owner);
        _;
    }

   //设置管理者owner地址
    constructor() public {
        owner = msg.sender;
    }

   //更新实现合约地址
    function upgradeTo(address _newImplementation) external onlyOwner {
        require(implementation != _newImplementation);
        _setImplementation(_newImplementation);
    }

   //回调
    function () payable public {
        address impl = implementation;
        require(impl != address(0));
        assembly {
            let ptr := mload(0x40)
            calldatacopy(ptr, 0, calldatasize)
            let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0)
            let size := returndatasize
            returndatacopy(ptr, 0, size)

            switch result
            case 0 { revert(ptr, size) }
            default { return(ptr, size) }
        }
    }

    //设置当前实现地址
    function _setImplementation(address _newImp) internal {
        implementation = _newImp;
    }
}
```

## 3.2 逻辑合约

1. 管理员的管理：管理员由合约owner管理；管理员和owner能查询所有链在合约内的信息，申请企业只能查看自己链在合约内的信息；成为管理员需要发起申请，通过业务平台由owner或者超级管理员中的任何一个审核(同意/删除/拒绝/暂时禁用，需附原因)；管理员可以发起申请成为超级管理员的请求，由owner审核(同意/删除/拒绝/暂时禁用，需附原因)；超级管理员可以管理普通管理员(同意/删除/拒绝/暂时禁用，需附原因)；
2. 为防止owner发生意外事件，导致合约运行受阻，采用owner的继承者来解决此问题。继承者由owner指定，但初始无任何权限，owner可随时更换/启用继承者，若继承者自发要求启用，需要继承者发起请求，由合约当前owner或者在规定时间内超过2/3的超级管理员对此请求进行审核，方能实现替换操作，请求时需要指定当前继承者，自己的继承者，即继承者的继承者。历代合约的onwer，都用一个map记录下来，以备不时之需。实现更换后，继承者才将拥有和owner一样的权限，代替之前的合约owner，成为新的owner。
3. 骨干链的管理：企业或单位发起申请注册骨干链请求；业务平台审核，通过合约中的方法同意/拒绝，同意后分配得到chiancode和chainid；管理员和owner有权力添加/删除/停用链；申请企业能停用自己的链；

3. 骨干链的子链管理：子链信息上骨干链，包括子链区块头信息、用户信息、节点信息、合约信息，提供信息注册、查询、更新的功能，并依据人员权限赋予不同的操作功能；

## 3.3 存储合约

合约的数据结构包括管理员、骨干链、子链三大部分；

管理员数据结构：

- manager_address 骨干链地址
- manager_name 名称

- manager_time 注册事件
- manager_signature 审核人

骨干链数据结构：

- chain_code 链的编码
- chain_id 链的id
- chain_address 链的地址
- chain_user 链的用户信息
- chain_blockheader 链的区块头信息
- chain_constract 链的合约信息
- chain_node 链的节点信息

子链数据结构：

1. 用户
    - name 姓名
    - bid 用户bid
    - email 电子邮件
    - address 联系地址
    - region 所在地区
    - mobile 手机号
2. 节点信息
    - name 子链名称
    - version 版本号
    - algorithm 共识算法
    - chain_code 链的编码
    - chain_id 链的ID
    - industry 行业
    - sort 子链分类
    - scene 场景描述
    - website 子链官网
    - browse 子链浏览器
    - base_chain 底层链名称
    - node_json 节点的json串
    - consensus_bid 共识节点bid
    - service_ip 地址IP
    - http_port HTTP端口
    - websocket_port Websocke端口
    - config_json 配置文件
    - link_man 联系人
    - link_phone 联系电话
    - logo 企业logo
    - owner_bid 所有者（企业）bid
    - origin_time 链创建时间
    - up_hash 上报的交易哈希
    - up_status 上报状态
    - up_time 上报时间
3. 合约信息
    - name 合约名称
    - address 合约地址
    - type 合约类型
    - owner_bid 创建人bid

- created 合约创建时间
- byte_code 字节码
- abi abi
- chain_code 链的名称
- chain_id 链的ID
- chain_name 链名称
- up_hash 上报的交易哈希
- up_status 上报状态
- up_time 上报时间

4. 区块头信息

- chain_code 链code
- chain_id 链id
- chain_name 链名称
- block_header_hash 区块头哈希
- transaction_num 交易数量
- block_man 出块人
- block_time 出块时间