基于 XuperChain 存证流程合约开发

·Caller.sol

```
pragma solidity >=0.4.22 <0.6.0;

import "./Ownable.sol";

contract Caller is Ownable {

  mapping(address => bool) public caller;

  uint256 public callerAmount;

  modifier isCaller() {

    require(caller[msg.sender] == true, "Caller is not grantor");

    _;

  }

   function authorize(address _user) public onlyOwner {

    caller[_user] = true;

    callerAmount++;

  }

  function deAuthorize(address _user) public onlyOwner {

    caller[_user] = false;

    callerAmount--;

  }

}
```

·EvidenceBaseSaveHandler.sol

```
pragma solidity >= 0.4 .22 < 0.6 .0;

import "./EvidenceData.sol";

import "./Ownable.sol";

contract EvidenceBaseSaveHandler is Ownable, EvidenceData {

  bool internal _initialized;/*是因为代理合约在代理逻辑合约之后，逻辑合约自身通过构造函数初始化的值是无法获取到的，

                因此需要有一个方法能够为初始参数赋值*/

  function initialize(address owner) public {

    require(!_initialized);

    setOwner(owner);
```

```
    _initialized = true;
  }
  function createSaveEvidence(bytes32 _hash, bytes memory _content) public onlyOwner {
    require(keccak256(_content) == _hash, "Invalid hash!");
    require(evidence[_hash].owner == address(0), "Evidence exist!");
    evidence[_hash] = EvidenceObject({
      content: _content,
      owner: msg.sender,
      timestamp: now
    });
    evidenceAmount++;
  }
  function getEvidence(bytes32 _hash) public view returns(bytes memory content, uint256
timestamp) {
    return (evidence[hash].content, evidence[hash].timestamp);
  }
  function checkEvidenceExist(bytes32 _hash) public view returns(bool isExist) {
    isExist = false;
    if (evidence[_hash].owner != address(0)) {
      isExist = true;
    }
    return isExist;
  }
  function getEvidenceAmount() public view returns(uint256 amount) {
    return evidenceAmount;
  }
}
```

·EvidenceData.sol

```
pragma solidity >=0.4.22 <0.6.0;
contract EvidenceData {
  struct EvidenceObject {
    bytes content;//存证内容
    address owner;//所有者
```

```
    uint timestamp;//存证时间
  }
  mapping(bytes32 => EvidenceObject) internal evidence;//存证hash与存证结构的mapping变量
  uint internal evidenceAmount;
}
```

·EvidenceVoteSaveHandler.sol

```
pragma solidity >= 0.4 .22 < 0.6 .0;
import "./EvidenceBaseSaveHandler.sol";
import "./SafeMath.sol";
import "./Caller.sol";
contract EvidenceVoteSaveHandler is EvidenceBaseSaveHandler, Caller {
  using SafeMath
  for uint256;
  struct VoteEvidenceObject {
    address owner;
    bytes content;
    uint8 voted; // 赞成票个数
    mapping(address => bool) voters; // 审核方投票记录
  }
  mapping(bytes32 => VoteEvidenceObject) private voteEvidence; // 存证方发起的存证 存储到待上链的
  uint8 public threshold; // 投票阈值，超过该阈值则说明存证内容可上链
  function setThreshold(uint8 _threshold) public isCaller {
    threshold = _threshold;
  }
  // 存证方发起存证，会先存储到待上链的voteEvidence中
  function createSaveEvidence(bytes32 _hash, bytes memory _content) public isCaller {
    require(keccak256(_content) == _hash, "Invalid hash!");
    require(voteEvidence[_hash].owner == address(0), "Vote evidence exist!");
    require(checkEvidenceExist(_hash) == false, "Evidence exist!");
    voteEvidence[_hash] = VoteEvidenceObject({
      content: _content,
      owner: msg.sender,
```

```
      voted: 0
    });
  }
  // 对待上链的存证进行投票
  function voteEvidenceToChain(bytes32 _hash) public isCaller {
    require(voteEvidence[_hash].owner != address(0), "Evidence not exist!");
    require(voteEvidence[_hash].voters[msg.sender] == false, "Already voted!");
    voteEvidence[_hash].voted++;
    voteEvidence[_hash].voters[msg.sender] = true;
  }
  // 对超过投票阈值的存证发起上链
  function saveEvidenceToChain(bytes32 _hash) public {
    require(voteEvidence[_hash].owner != address(0), "Evidence not exist!");
    require(checkEvidenceExist(_hash) == false, "Evidence exist!");
    require(uint256(voteEvidence[_hash].voted).mul(100).div(callerAmount) >= threshold,
"Insufficient votes!");
    evidence[_hash] = EvidenceObject({
      content: voteEvidence[_hash].content,
      owner: msg.sender,
      timestamp: now
    });
    evidenceAmount++;
  }
}
```

·Ownable.sol

```
pragma solidity >= 0.4 .22 < 0.6 .0;
 * @title Ownable
 * @dev 此合约具有提供基本授权控制的所有者地址
 */
contract Ownable {
 /**
  * @dev 显示所有权的事件已转让
  * @上一个表示前所有者地址的所有者
```

```
 * @param 新所有者 表示新所有者的地址
 */
event OwnershipTransferred(address previousOwner, address newOwner);
// Owner of the contract
address private _owner;
/**
 * @dev 如果由所有者以外的任何帐户调用，则抛出。
 */
modifier onlyOwner() {
 require(msg.sender == owner());
 _;
}
/**
 * @dev 构造函数将合约的原始所有者设置为发送方帐户。
 */
constructor() public {
 setOwner(msg.sender);
}
/**
 * @开发人员 告诉所有者的地址
 * @返回所有者的地址
 */
function owner() public view returns (address) {
 return _owner;
}
/**
 * @开发人员 设置新的所有者地址
 */
function setOwner(address newOwner) internal {
 _owner = newOwner;
}
/**
 *@开发人员 允许当前所有者将合同的控制权转移给新所有者。
 * @param新所有者 要将所有权转让给的地址。
```

```
 */
 function transferOwnership(address newOwner) public onlyOwner {
  require(newOwner != address(0));
  emit OwnershipTransferred(owner(), newOwner);
  setOwner(newOwner);
 }
}
```

·OwnedUpgradeabilityProxy.sol

```
pragma solidity ^0.4.21;
import './UpgradeabilityProxy.sol';
/**
 * @title 拥有的可升级性代理
 * @dev 此合约将可升级性代理与基本授权控制功能相结合
 */
contract OwnedUpgradeabilityProxy is UpgradeabilityProxy {
 /**
 * @dev显示所有权的事件已转让
 * @param 上一个代表前所有者地址的所有者
 * @param 新所有者代表新所有者的地址
 */
 event ProxyOwnershipTransferred(address previousOwner, address newOwner);
 // 合同所有者的存储位置
 bytes32 private constant proxyOwnerPosition = keccak256("org.zeppelinos.proxy.owner");
 /**
 * @dev 构造函数将合约的原始所有者设置为发送方帐户。
 */
 function OwnedUpgradeabilityProxy() public {
  setUpgradeabilityOwner(msg.sender);
 }
 /**
 * @dev 如果由所有者以外的任何帐户调用，则抛出。
 */
 modifier onlyProxyOwner() {
```

```
 require(msg.sender == proxyOwner());
 _;
}
/**
 * @告知所有者的地址
 * @return所有者的地址
 */
function proxyOwner() public view returns (address owner) {
 bytes32 position = proxyOwnerPosition;
 assembly {
  owner := sload(position)
 }
}
/**
 * @dev 设置所有者的地址
 */
function setUpgradeabilityOwner(address newProxyOwner) internal {
 bytes32 position = proxyOwnerPosition;
 assembly {
  sstore(position, newProxyOwner)
 }
}
/**
 * @dev 允许当前所有者将合同的控制权转移给新所有者。
 * @param新所有者 要将所有权转让给的地址。
 */
function transferProxyOwnership(address newOwner) public onlyProxyOwner {
 require(newOwner != address(0));
 emit ProxyOwnershipTransferred(proxyOwner(), newOwner);
 setUpgradeabilityOwner(newOwner);
}
/**
 * @dev Allows the proxy owner to upgrade the current version of the proxy.
 * @param implementation representing the address of the new implementation to be set.
```

```
 */
 function upgradeTo(address implementation) public onlyProxyOwner {
  _upgradeTo(implementation);
 }
 /**
  * @dev Allows the proxy owner to upgrade the current version of the proxy and call the new implementation
  * to initialize whatever is needed through a low level call.
  * @param implementation representing the address of the new implementation to be set.
  * @param data represents the msg.data to bet sent in the low level call. This parameter may include the function
  * signature of the implementation to be called with the needed payload
  */
 function upgradeToAndCall(address implementation, bytes data) payable public onlyProxyOwner {
  upgradeTo(implementation);
  require(this.call.value(msg.value)(data));
 }
}
```

·Proxy.sol

```
pragma solidity ^0.4.21;
/**
 * @title Proxy
 * @dev 提供了将任何调用委托给外部实现的可能性。
 */
contract Proxy {
 /**
 * @dev 告知将委派每个调用的实现的地址。
 * @return 将委派给的实现的地址
 function implementation() public view returns (address);
 /**
 * 回退函数允许对给定实现执行委托调用。
 * 此函数将返回实现调用返回的任何内容
 */
```

```
 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) }
  }
 }
}
```

·UpgradeabilityProxy.sol

```
pragma solidity ^0.4.21;
import './Proxy.sol';
/**
 * @title可升级性代理
 * @dev 此协定表示一个代理，其中可以升级它将委派到的实现地址
 */
contract UpgradeabilityProxy is Proxy {
 /**
  * @dev 每次升级实现时都会发出此事件
  * @param实现，表示升级后实现的地址
  */
 event Upgraded(address indexed implementation);
 // 当前实现地址的存储位置
 bytes32 private constant implementationPosition =
keccak256("org.zeppelinos.proxy.implementation");
 /**
  * @dev 构造函数
```

```
 */
function UpgradeabilityProxy() public {}
/**
 * @dev 告诉当前实现的地址
 * 当前实施的@return地址
 */
function implementation() public view returns (address impl) {
 bytes32 position = implementationPosition;
 assembly {
  impl := sload(position)
 }
}
/**
 * @dev 设置当前实现的地址
 * @param表示要设置的新实现的新实现地址
 */
function setImplementation(address newImplementation) internal {
 bytes32 position = implementationPosition;
 assembly {
  sstore(position, newImplementation)
 }
}
/**
 * @dev 升级实施地址
 * @param新实现，表示要设置的新实现的地址
 */
function _upgradeTo(address newImplementation) internal {
 address currentImplementation = implementation();
 require(currentImplementation != newImplementation);
 setImplementation(newImplementation);
 emit Upgraded(newImplementation);
}
}
```