知道创宇

密级 ★ | 项目组公开

智能合约审计报告

安全状态

安全

★ ★ ★ ★ ★

主测人： 知道创宇区块链安全研究团队

版本说明

修订内容	时间	修订者	版本号
编写文档	20210813	知道创宇区块链安全研究团队	V1.0

文档信息

文档名称	文档版本	报告编号	保密级别
CAN 智能合约审计报告	V1.0	c51b79a8829a4f078f8610d6fabb6953	项目组公开

声明

创宇仅就本报告出具前已经发生或存在的事实出具本报告，并就此承担相应责任。对于出具以后发生或存在的事实，创宇无法判断其智能合约安全状况，亦不对此承担责任。本报告所作的安全审计分析及其他内容，仅基于信息提供者截至本报告出具时向创宇提供的文件和资料。创宇假设:已提供资料不存在缺失、被篡改、删减或隐瞒的情形。如已提供资料信息缺失、被篡改、删减、隐瞒或反映的情况与实际情况不符的，创宇对由此而导致的损失和不利影响不承担任何责任。

目录

1. 综述

本次报告有效测试时间是从 2021 年 8 月 11 日开始到 2021 年 8 月 13 日结束，在此期间针对 **CAN 智能合约代币代码及兑换矿池代码**的安全性和规范性进行审计并以此作为报告统计依据。

本次智能合约安全审计的范围，不包括外部合约调用，不包含未来可能出现的新型攻击方式，不包含合约升级或篡改后的代码（随着项目方的发展，智能合约可能会增加新的 pool、新的功能模块，新的外部合约调用等），不包含前端安全与服务器安全。

此次测试中，知道创宇工程师对智能合约的常见漏洞（见第三章节）进行了全面的分析，综合评定为通过。

本次智能合约安全审计结果： 通过

由于本次测试过程在非生产环境下进行，所有代码均为最新备份，测试过程均与相关接口人进行沟通，并在操作风险可控的情况下进行相关测试操作，以规避测试过程中的生产运营风险、代码安全风险。

本次审计的报告信息：

报告编号：c51b79a8829a4f078f8610d6fabb6953

报告查询地址链接:

https://attest.im/attestation/searchResult?qurey=c51b79a8829a4f078f8610d6fabb6953

本次审计的目标信息：

条目	描述
项目名称	Channels
Token 名称	CAN
合约地址	0xd5F9bdc2e6c8EE0484a6293ce7FA97d96a5e1012
代码类型	代币合约、DeFi 协议代码

代码语言	Solidity

合约文件及哈希：

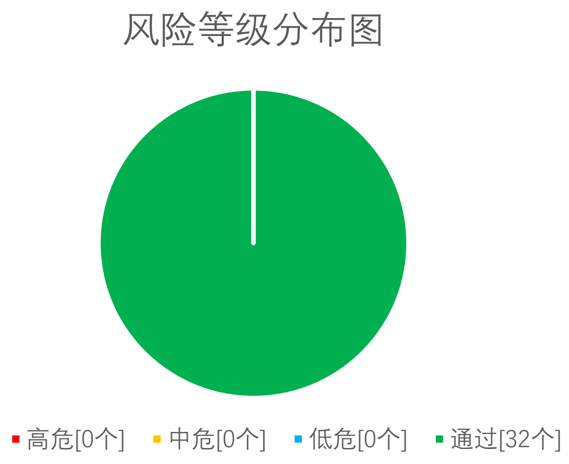
合约文件	MD5
Can.sol	b1484a766a9cba6f080a5a6cf5c72fa0
Comptroller.sol	7f5aba4a0701ebda3af402a480a49f0b
ExchangePool.sol	4802ac34b9371982e33c1d472e0ac58d
Unitroller.sol	d18cd0ec7a8b5dcf4b2c3c2abc2951c0

2. 代码漏洞分析

2.1 漏洞等级分布

本次漏洞风险按等级统计：

安全风险等级个数统计表			
高危	中危	低危	通过
0	0	0	32



2.2 审计结果汇总说明

审计结果			
审计项目	审计内容	状态	描述
业务安全性检测	Can 合约代币功能	通过	经检测，不存在安全问题。
	ExchangePool 合约代币以旧换新功能	通过	经检测，不存在安全问题。
	Comptroller 合约铸币权	通过	经检测，不存在安全问题。
	Comptroller 合约设置应结算元素	通过	经检测，不存在安全问题。
	Unitroller 合约更换 Admin 功能	通过	经检测，不存在安全问题。
代码基本漏洞检测	编译器版本安全	通过	经检测，不存在该安全问题。
	冗余代码	通过	经检测，不存在该安全问题。
	安全算数库的使用	通过	经检测，不存在该安全问题。
	不推荐的编码方式	通过	经检测，不存在该安全问题。
	require/assert 的合理使用	通过	经检测，不存在该安全问题。
	fallback 函数安全	通过	经检测，不存在该安全问题。
	tx.orgin 身份验证	通过	经检测，不存在该安全问题。
	owner 权限控制	通过	经检测，不存在该安全问题。
	gas 消耗检测	通过	经检测，不存在该安全问题。
	call 注入攻击	通过	经检测，不存在该安全问题。
	低级函数安全	通过	经检测，不存在该安全问题。
	增发代币漏洞	通过	经检测，不存在该安全问题。
	访问控制缺陷检测	通过	经检测，不存在该安全问题。
	数值溢出检测	通过	经检测，不存在该安全问题。

	算数精度误差	通过	经检测，不存在该安全问题。
	错误使用随机数检测	通过	经检测，不存在该安全问题。
	不安全的接口使用	通过	经检测，不存在该安全问题。
	变量覆盖	通过	经检测，不存在该安全问题。
	未初始化的存储指针	通过	经检测，不存在该安全问题。
	返回值调用验证	通过	经检测，不存在该安全问题。
	交易顺序依赖检测	通过	经检测，不存在该安全问题。
	时间戳依赖攻击	通过	经检测，不存在该安全问题。
	拒绝服务攻击检测	通过	经检测，不存在该安全问题。
	假充值漏洞检测	通过	经检测，不存在该安全问题。
	重入攻击检测	通过	经检测，不存在该安全问题。
	重放攻击检测	通过	经检测，不存在该安全问题。
	重排攻击检测	通过	经检测，不存在该安全问题。

3. 业务安全性检测

3.1. Can 合约代币功能【通过】

审计分析：代币功能在 Can 合约文件中实现，符合基础的代币实现标准。

```
contract Can {
    /// @notice EIP-20 token name for this token
    string public constant name = "Channels";// knownsec //代币名称

    /// @notice EIP-20 token symbol for this token
    string public constant symbol = "CAN";// knownsec //代币简称

    /// @notice EIP-20 token decimals for this token
    uint8 public constant decimals = 18;// knownsec //代币精度

    /// @notice Total number of tokens in circulation
    uint public constant totalSupply = 1000000000e18; // todo: update from 10 million to 1 bil
lion Can // knownsec //发行总量

    /// @notice Allowance amounts on behalf of others
    mapping (address => mapping (address => uint96)) internal allowances;

    /// @notice Official record of token balances for each account
    mapping (address => uint96) internal balances;

    /// @notice A record of each accounts delegate
    mapping (address => address) public delegates;

    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint96 votes;
    }

    /// @notice A record of votes checkpoints for each account, by index
    mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;

    /// @notice The number of checkpoints for each account
    mapping (address => uint32) public numCheckpoints;

    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name
,uint256 chainId,address verifyingContract)");

    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address de
legatee,uint256 nonce,uint256 expiry)");

    /// @notice A record of states for signing / validating signatures
    mapping (address => uint) public nonces;

    /// @notice An event thats emitted when an account changes its delegate
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, addres
s indexed toDelegate);

    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBal
ance);

    /// @notice The standard EIP-20 transfer event
    event Transfer(address indexed from, address indexed to, uint256 amount);

    /// @notice The standard EIP-20 approval event
    event Approval(address indexed owner, address indexed spender, uint256 amount);
```

```
/**
 * @notice Construct a new Can token
 * @param account The initial account to grant all the tokensgnrh
 */
constructor(address account) public {
   balances[account] = uint96(totalSupply);
   emit Transfer(address(0), account, totalSupply);
}
```

安全建议：无。

3.2. ExchangePool 合约代币以旧换新功能【通过】

审计分析：代币以旧换新功能由 exchange 函数实现，主要逻辑先检查兑换地址代币数量，在通过 safeTransferFrom 与 safeTransfer 函数实现新旧代币兑换。经审计，功能逻辑正确，权限校验合理。

```
function exchange(uint amount) public checkOpen lock{//knownsec //将旧币兑换成新币
   require(preCAN.balanceOf(msg.sender) >= amount, "preCAN not enough");
   preCAN.safeTransferFrom(msg.sender, address(this), amount);
   newCAN.safeTransfer(msg.sender, amount.mul(200));
   emit Exchange(msg.sender, amount, amount.mul(200));
}
```

安全建议：无。

3.3. Comptroller 合约铸币权【通过】

审计分析：铸币权功能由 Comptroller 合约文件中 mintAllowed 函数实现，其主要逻辑是先检查铸币功能是否开放，再判断被授权地址是否是成员地址，不

是则返回错误，正确则更新 Can 代币发行指数，记录被授权地址分配权限。经审计，功能逻辑正确，权限校验合理。

```
    function mintAllowed(address cToken, address minter, uint mintAmount) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!mintGuardianPaused[cToken], "mint is paused");

        // Shh - currently unused
        minter;
        mintAmount;

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // Keep the flywheel moving
        updateCanSupplyIndex(cToken);
        distributeSupplierCan(cToken, minter, false);

        return uint(Error.NO_ERROR);
    }// knownsec 铸币权
```

安全建议：无。

3.4. Comptroller 合约设置应结算元素【通过】

审计分析：设置应结算元素功能由_setCloseFactor 函数实现，其逻辑先判断调用者是否为 admin，再检查其指数区间是否正确，无误则实施更换。经审计，功能逻辑正确，权限校验合理。

```
  function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint) {
    // Check caller is admin
    if (msg.sender != admin) {
      return fail(Error.UNAUTHORIZED, FailureInfo.SET_CLOSE_FACTOR_OWNER_CHEC
K);
    }

    Exp memory newCloseFactorExp = Exp({mantissa: newCloseFactorMantissa});
    Exp memory lowLimit = Exp({mantissa: closeFactorMinMantissa});
    if (lessThanOrEqualExp(newCloseFactorExp, lowLimit)) {
      return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALID
ATION);
    }

    Exp memory highLimit = Exp({mantissa: closeFactorMaxMantissa});
    if (lessThanExp(highLimit, newCloseFactorExp)) {
      return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALID
ATION);
    }

    uint oldCloseFactorMantissa = closeFactorMantissa;
    closeFactorMantissa = newCloseFactorMantissa;
    emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);

    return uint(Error.NO_ERROR);
  }// knownsec 设置应结算元素
```

安全建议：无。

3.5. Unitroller 合约更换 Admin 功能【通过】

审计分析：Unitroller 合约使用_setPendingAdmin 和_acceptAdmin 函数来进

行更换 Admin，_setPendingAdmin 函数负责确定候选人，_acceptAdmin 函数确定继承者。经审计，功能逻辑正确，权限校验合理。

```
    function _setPendingAdmin(address newPendingAdmin) public returns (uint) {
        // Check caller = admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_ADMIN_OWNER_CHECK);
        }

        // Save current value, if any, for inclusion in log
        address oldPendingAdmin = pendingAdmin;

        // Store pendingAdmin with value newPendingAdmin
        pendingAdmin = newPendingAdmin;

        // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
        emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

        return uint(Error.NO_ERROR);
    }// knownsec 设置候选 Admin

    /**
     * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
     * @dev Admin function for pending admin to accept role and update admin
     * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
     */
    function _acceptAdmin() public returns (uint) {
        // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
        if (msg.sender != pendingAdmin || msg.sender == address(0)) {
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK);
        }
        // Save current values for inclusion in log
        address oldAdmin = admin;
        address oldPendingAdmin = pendingAdmin;

        // Store admin with value pendingAdmin
        admin = pendingAdmin;

        // Clear the pending value
        pendingAdmin = address(0);

        emit NewAdmin(oldAdmin, admin);
        emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);

        return uint(Error.NO_ERROR);
    }// knownsec 确定新 Admin
```

安全建议：无。

4. 代码基本漏洞检测

4.1. 编译器版本安全【通过】

检查合约代码实现中是否使用了安全的编译器版本

检测结果：经检测，智能合约代码中制定了编译器版本大于等于 0.5.16，不存在该安全问题。

安全建议：无。

4.2. 冗余代码【通过】

检查合约代码实现中是否包含冗余代码

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.3. 安全算数库的使用【通过】

检查合约代码实现中是否使用了 SafeMath 安全算数库

检测结果：经检测，智能合约代码中已使用 SafeMath 安全算数库，不存在该安全问题。

安全建议：无。

4.4. 不推荐的编码方式【通过】

检查合约代码实现中是否有官方不推荐或弃用的编码方式

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.5. require/assert 的合理使用【通过】

检查合约代码实现中 require 和 assert 语句使用的合理性

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.6. fallback 函数安全【通过】

检查合约代码实现中是否正确使用 fallback 函数

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.7. tx.origin 身份验证【通过】

tx.origin 是 Solidity 的一个全局变量，它遍历整个调用栈并返回最初发送调用（或事务）的帐户的地址。在智能合约中使用此变量进行身份验证会使合约容易受到类似网络钓鱼的攻击。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.8. owner 权限控制【通过】

检查合约代码实现中的 owner 是否具有过高的权限。例如，任意修改其他账户余额等。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.9. gas 消耗检测【通过】

检查 gas 的消耗是否超过区块最大限制

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.10. call 注入攻击【通过】

call 函数调用时，应该做严格的权限控制，或直接写死 call 调用的函数。

检测结果：经检测，智能合约不存在此漏洞。

安全建议：无。

4.11. 低级函数安全【通过】

检查合约代码实现中低级函数（call/delegatecall）的使用是否存在安全漏洞

call 函数的执行上下文是在被调用的合约中；而 delegatecall 函数的执行上下文是在当前调用该函数的合约中

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.12. 增发代币漏洞【通过】

检查在初始化代币总量后，代币合约中是否存在可能使代币总量增加的函数。

检测结果： 经检测，智能合约代码中不存在该安全问题。

安全建议： 无。

4.13. 访问控制缺陷检测【通过】

合约中不同函数应设置合理的权限

检查合约中各函数是否正确使用了 public、private 等关键词进行可见性修饰，检查合约是否正确定义并使用了 modifier 对关键函数进行访问限制，避免越权导致的问题。

检测结果： 经检测，智能合约代码中不存在该安全问题。

安全建议： 无。

4.14. 数值溢出检测【通过】

智能合约中的算数问题是指整数溢出和整数下溢。

Solidity 最多能处理 256 位的数字（2^256-1），最大数字增加 1 会溢出得到 0。同样，当数字为无符号类型时，0 减去 1 会下溢得到最大数字值。

整数溢出和下溢不是一种新类型的漏洞，但它们在智能合约中尤其危险。溢出情况会导致不正确的结果，特别是如果可能性未被预期，可能会影响程序的可靠性和安全性。

检测结果： 经检测，智能合约代码中不存在该安全问题。

安全建议： 无。

4.15. 算术精度误差【通过】

Solidity 作为一门编程语言具备和普通编程语言相似的数据结构设计，比如：变量、常量、函数、数组、函数、结构体等等，Solidity 和普通编程语言也有一个较大的区别——Solidity 没有浮点型，且 Solidity 所有的数值运算结果都只会是整数，不会出现小数的情况，同时也不允许定义小数类型数据。合约中的数值运算必不可少，而数值运算的设计有可能造成相对误差，例如同级运算：5/2*10=20，而 5*10/2=25，从而产生误差，在数据更大时产生的误差也会更大，更明显。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.16. 错误使用随机数【通过】

智能合约中可能需要使用随机数，虽然 Solidity 提供的函数和变量可以访问明显难以预测的值，如 block.number 和 block.timestamp，但是它们通常或者比看起来更公开，或者受到矿工的影响，即这些随机数在一定程度上是可预测的，所以恶意用户通常可以复制它并依靠其不可预知性来攻击该功能。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.17. 不安全的接口使用【通过】

检查合约代码实现中是否使用了不安全的接口

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.18. 变量覆盖【通过】

检查合约代码实现中是否存在变量覆盖导致的安全问题

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.19. 未初始化的储存指针【通过】

在 solidity 中允许一个特殊的数据结构为 struct 结构体，而函数内的局部变量默认使用 storage 或 memory 储存。

而存在 storage(存储器)和 memory(内存)是两个不同的概念，solidity 允许指针指向一个未初始化的引用，而未初始化的局部 stroage 会导致变量指向其他储存变量，导致变量覆盖，甚至其他更严重的后果，在开发中应该避免在函数中初始化 struct 变量。

检测结果：经检测，智能合约代码不存在该问题。

安全建议：无。

4.20. 返回值调用验证【通过】

此问题多出现在和转币相关的智能合约中，故又称作静默失败发送或未经检查发送。

在 Solidity 中存在 transfer()、send()、call.value()等转币方法，都可以用于向某一地址发送 HT，其区别在于：transfer 发送失败时会 throw，并且进行状态回滚；只会传递 2300gas 供调用，防止重入攻击；send 发送失败时会返回 false；只会传递 2300gas 供调用，防止重入攻击；call.value 发送失败时会返回 false；

传递所有可用 gas 进行调用（可通过传入 gas_value 参数进行限制），不能有效防止重入攻击。

如果在代码中没有检查以上 send 和 call.value 转币函数的返回值，合约会继续执行后面的代码，可能由于 HT 发送失败而导致意外的结果。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.21. 交易顺序依赖【通过】

由于矿工总是通过代表外部拥有地址（EOA）的代码获取 gas 费用，因此用户可以指定更高的费用以便更快地开展交易。由于以太坊区块链是公开的，每个人都可以看到其他人未决交易的内容。这意味着，如果某个用户提交了一个有价值的解决方案，恶意用户可以窃取该解决方案并以较高的费用复制其交易，以抢占原始解决方案。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.22. 时间戳依赖攻击【通过】

数据块的时间戳通常来说都是使用矿工的本地时间，而这个时间大约能有 900 秒的范围波动，当其他节点接受一个新区块时，只需要验证时间戳是否晚于之前的区块并且与本地时间误差在 900 秒以内。一个矿工可以通过设置区块的时间戳来尽可能满足有利于他的条件来从中获利。

检查合约代码实现中是否存在有依赖于时间戳的关键功能

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.23. 拒绝服务攻击【通过】

在以太坊的世界中，拒绝服务是致命的，遭受该类型攻击的智能合约可能永远无法恢复正常工作状态。导致智能合约拒绝服务的原因可能有很多种，包括在作为交易接收方时的恶意行为，人为增加计算功能所需 gas 导致 gas 耗尽，滥用访问控制访问智能合约的 private 组件，利用混淆和疏忽等等。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.24. 假充值漏洞【通过】

在代币合约的 transfer 函数对转账发起人(msg.sender)的余额检查用的是 if 判断方式，当 balances[msg.sender] < value 时进入 else 逻辑部分并 return false，最终没有抛出异常，我们认为仅 if/else 这种温和的判断方式在 transfer 这类敏感函数场景中是一种不严谨的编码方式。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.25. 重入攻击检测【通过】

重入漏洞是最著名的以太坊智能合约漏洞，曾导致了以太坊的分叉（The DAO hack）。

Solidity 中的 call.value()函数在被用来发送 HT 的时候会消耗它接收到的所有 gas，当调用 call.value()函数发送 HT 的操作发生在实际减少发送者账户的余额之前时，就会存在重入攻击的风险。

检测结果：经检测，智能合约代码中不存在该安全问题。

安全建议：无。

4.26. 重放攻击检测【通过】

合约中如果涉及委托管理的需求，应注意验证的不可复用性，避免重放攻击

在资产管理体系中，常有委托管理的情况，委托人将资产给受托人管理，委托人支付一定的费用给受托人。这个业务场景在智能合约中也比较普遍。。

检测结果：经检测，智能合约不存在此漏洞。

安全建议：无。

4.27. 重排攻击检测【通过】

重排攻击是指矿工或其他方试图通过将自己的信息插入列表(list)或映射(mapping)中来与智能合约参与者进行“竞争”，从而使攻击者有机会将自己的信息存储到合约中。

检测结果:经检测，智能合约代码中不存在相关漏洞。

安全建议:无。

5. 附录 A：合约代码

本次测试代码来源:

Wwrp.sol

```
pragma solidity ^0.5.16;
pragma experimental ABIEncoderV2;

contract Can {
	/// @notice EIP-20 token name for this token
	string public constant name = "Channels";// knownsec //代币名称

	/// @notice EIP-20 token symbol for this token
	string public constant symbol = "CAN";// knownsec //代币简称

	/// @notice EIP-20 token decimals for this token
	uint8 public constant decimals = 18;// knownsec //代币精度

	/// @notice Total number of tokens in circulation
	uint public constant totalSupply = 1000000000e18; // todo: update from 10 million to 1 billion Can // knownsec
//发行总量

	/// @notice Allowance amounts on behalf of others
	mapping (address => mapping (address => uint96)) internal allowances;

	/// @notice Official record of token balances for each account
	mapping (address => uint96) internal balances;

	/// @notice A record of each accounts delegate
	mapping (address => address) public delegates;

	/// @notice A checkpoint for marking number of votes from a given block
	struct Checkpoint {
		uint32 fromBlock;
		uint96 votes;
	}

	/// @notice A record of votes checkpoints for each account, by index
	mapping (address => mapping (uint32 => Checkpoint)) public checkpoints;

	/// @notice The number of checkpoints for each account
	mapping (address => uint32) public numCheckpoints;

	/// @notice The EIP-712 typehash for the contract's domain
	bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256
chainId,address verifyingContract)");

	/// @notice The EIP-712 typehash for the delegation struct used by the contract
	bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256
nonce,uint256 expiry)");

	/// @notice A record of states for signing / validating signatures
	mapping (address => uint) public nonces;

	/// @notice An event thats emitted when an account changes its delegate
	event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed
toDelegate);

	/// @notice An event thats emitted when a delegate account's vote balance changes
	event DelegateVotesChanged(address indexed delegate, uint previousBalance, uint newBalance);

	/// @notice The standard EIP-20 transfer event
	event Transfer(address indexed from, address indexed to, uint256 amount);

	/// @notice The standard EIP-20 approval event
	event Approval(address indexed owner, address indexed spender, uint256 amount);

	/**
	 * @notice Construct a new Can token
	 * @param account The initial account to grant all the tokensgnrh
	 */
	constructor(address account) public {
		balances[account] = uint96(totalSupply);
		emit Transfer(address(0), account, totalSupply);
	}

	/**
	 * @notice Get the number of tokens `spender` is approved to spend on behalf of `account`
	 * @param account The address of the account holding the funds
	 * @param spender The address of the account spending the funds
	 * @return The number of tokens approved
```

```
     */
    function allowance(address account, address spender) external view returns (uint) {
        return allowances[account][spender];
    }

    /**
     * @notice Approve `spender` to transfer up to `amount` from `src`
     * @dev This will overwrite the approval amount for `spender`
     *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
     * @param spender The address of the account which may transfer tokens
     * @param rawAmount The number of tokens that are approved (2^256-1 means infinite)
     * @return Whether or not the approval succeeded
     */
    function approve(address spender, uint rawAmount) external returns (bool) {//knownsec //授权
        uint96 amount;
        if (rawAmount == uint(-1)) {
            amount = uint96(-1);
        } else {
            amount = safe96(rawAmount, "Can::approve: amount exceeds 96 bits");
        }

        allowances[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
        return true;
    }

    /**
     * @notice Get the number of tokens held by the `account`
     * @param account The address of the account to get the balance of
     * @return The number of tokens held
     */
    function balanceOf(address account) external view returns (uint) {
        return balances[account];
    }

    /**
     * @notice Transfer `amount` tokens from `msg.sender` to `dst`
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transfer(address dst, uint rawAmount) external returns (bool) {
        uint96 amount = safe96(rawAmount, "Can::transfer: amount exceeds 96 bits");
        _transferTokens(msg.sender, dst, amount);
        return true;
    }

    /**
     * @notice Transfer `amount` tokens from `src` to `dst`
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferFrom(address src, address dst, uint rawAmount) external returns (bool) {
        address spender = msg.sender;
        uint96 spenderAllowance = allowances[src][spender];
        uint96 amount = safe96(rawAmount, "Can::approve: amount exceeds 96 bits");

        if (spender != src && spenderAllowance != uint96(-1)) {
            uint96 newAllowance = sub96(spenderAllowance, amount, "Can::transferFrom: transfer amount
exceeds spender allowance");
            allowances[src][spender] = newAllowance;

            emit Approval(src, spender, newAllowance);
        }

        _transferTokens(src, dst, amount);
        return true;
    }

    /**
     * @notice Delegate votes from `msg.sender` to `delegatee`
     * @param delegatee The address to delegate votes to
     */
    function delegate(address delegatee) public {
        return _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to `delegatee`
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(address delegatee, uint nonce, uint expiry, uint8 v, bytes32 r, bytes32 s) public
{//knownsec //委托人签名
        bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name)),
getChainId(), address(this)));
        bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, expiry));
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "Can::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "Can::delegateBySig: invalid nonce");
        require(now <= expiry, "Can::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for `account`
     * @param account The address to get votes balance
     * @return The number of current votes for `account`
     */
    function getCurrentVotes(address account) external view returns (uint96) {//knownsec //获取账号所在区块的
当前投票数
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }

    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent misinformation.
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint blockNumber) public view returns (uint96) {//knownsec //获取账
号所在区块之前的投票数
        require(blockNumber < block.number, "Can::getPriorVotes: not yet determined");

        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {
            return 0;
        }

        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
            return checkpoints[account][nCheckpoints - 1].votes;
        }

        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) {
            return 0;
        }

        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee) internal {//knownsec //变更委托人
        address currentDelegate = delegates[delegator];
        uint96 delegatorBalance = balances[delegator];
        delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _transferTokens(address src, address dst, uint96 amount) internal {
        require(src != address(0), "Can::_transferTokens: cannot transfer from the zero address");
        require(dst != address(0), "Can::_transferTokens: cannot transfer to the zero address");

        balances[src] = sub96(balances[src], amount, "Can::_transferTokens: transfer amount exceeds
balance");
        balances[dst] = add96(balances[dst], amount, "Can::_transferTokens: transfer amount overflows");
        emit Transfer(src, dst, amount);
```

```
            _moveDelegates(delegates[src], delegates[dst], amount);
        }

        function _moveDelegates(address srcRep, address dstRep, uint96 amount) internal {
            if (srcRep != dstRep && amount > 0) {
                if (srcRep != address(0)) {
                    uint32 srcRepNum = numCheckpoints[srcRep];
                    uint96 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                    uint96 srcRepNew = sub96(srcRepOld, amount, "Can::_moveVotes: vote amount underflows");
                    _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
                }

                if (dstRep != address(0)) {
                    uint32 dstRepNum = numCheckpoints[dstRep];
                    uint96 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                    uint96 dstRepNew = add96(dstRepOld, amount, "Can::_moveVotes: vote amount overflows");
                    _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
                }
            }
        }

        function _writeCheckpoint(address delegatee, uint32 nCheckpoints, uint96 oldVotes, uint96 newVotes) internal
{
          uint32 blockNumber = safe32(block.number, "Can::_writeCheckpoint: block number exceeds 32 bits");

          if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
              checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
          } else {
              checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
              numCheckpoints[delegatee] = nCheckpoints + 1;
          }

          emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
        }

        function safe32(uint n, string memory errorMessage) internal pure returns (uint32) {
            require(n < 2**32, errorMessage);
            return uint32(n);
        }

        function safe96(uint n, string memory errorMessage) internal pure returns (uint96) {
            require(n < 2**96, errorMessage);
            return uint96(n);
        }

        function add96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96) {
            uint96 c = a + b;
            require(c >= a, errorMessage);
            return c;
        }

        function sub96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96) {
            require(b <= a, errorMessage);
            return a - b;
        }

        function getChainId() internal pure returns (uint) {
            uint256 chainId;
            assembly { chainId := chainid() }
            return chainId;
        }
}
```

Comptroller.sol

```
pragma solidity ^0.5.16;

import "./CToken.sol";
import "./ErrorReporter.sol";
import "./Exponential.sol";
import "./PriceOracle.sol";
import "./ComptrollerInterface.sol";
import "./ComptrollerStorage.sol";
import "./Unitroller.sol";
import "./Can.sol";
import "./SafeMath.sol";

/**
  * @title Channels's Comptroller Contract
  * @author Channels
  */
contract Comptroller is ComptrollerV3Storage, ComptrollerInterface, ComptrollerErrorReporter, Exponential {

        using SafeMath for *;

        /// @notice Emitted when an admin supports a market
        event MarketListed(CToken cToken);

        /// @notice Emitted when an account enters a market
```

```
    event MarketEntered(CToken cToken, address account);

    /// @notice Emitted when an account exits a market
    event MarketExited(CToken cToken, address account);

    /// @notice Emitted when close factor is changed by admin
    event NewCloseFactor(uint oldCloseFactorMantissa, uint newCloseFactorMantissa);

    /// @notice Emitted when a collateral factor is changed by admin
    event NewCollateralFactor(CToken cToken, uint oldCollateralFactorMantissa, uint
newCollateralFactorMantissa);

    /// @notice Emitted when liquidation incentive is changed by admin
    event NewLiquidationIncentive(uint oldLiquidationIncentiveMantissa, uint
newLiquidationIncentiveMantissa);

    /// @notice Emitted when maxAssets is changed by admin
    event NewMaxAssets(uint oldMaxAssets, uint newMaxAssets);

    /// @notice Emitted when price oracle is changed
    event NewPriceOracle(PriceOracle oldPriceOracle, PriceOracle newPriceOracle);

    /// @notice Emitted when pause guardian is changed
    event NewPauseGuardian(address oldPauseGuardian, address newPauseGuardian);

    /// @notice Emitted when an action is paused globally
    event ActionPaused(string action, bool pauseState);

    /// @notice Emitted when an action is paused on a market
    event ActionPaused(CToken cToken, string action, bool pauseState);

    /// @notice Emitted when market caned status is changed
    event MarketCaned(CToken cToken, bool isCaned);


    /// @notice Emitted when a new Channels speed is calculated for a market
    event CanSpeedUpdated(CToken indexed cToken, uint newSpeed);

    /// @notice Emitted when Channels is distributed to a supplier
    event DistributedSupplierCan(CToken indexed cToken, address indexed supplier, uint canDelta, uint
canSupplyIndex);

    /// @notice Emitted when Channels is distributed to a borrower
    event DistributedBorrowerCan(CToken indexed cToken, address indexed borrower, uint canDelta, uint
canBorrowIndex);


    /// @notice The threshold above which the flywheel transfers Channels, in wei
    uint public constant canClaimThreshold = 0.001e18;

    /// @notice The initial Channels index for a market
    uint224 public constant canInitialIndex = 1e36;

    // closeFactorMantissa must be strictly greater than this value
    uint internal constant closeFactorMinMantissa = 0.05e18; // 0.05

    // closeFactorMantissa must not exceed this value
    uint internal constant closeFactorMaxMantissa = 0.9e18; // 0.9

    // No collateralFactorMantissa may exceed this value
    uint internal constant collateralFactorMaxMantissa = 0.9e18; // 0.9

    // liquidationIncentiveMantissa must be no less than this value
    uint internal constant liquidationIncentiveMinMantissa = 1.0e18; // 1.0

    // liquidationIncentiveMantissa must be no greater than this value
    uint internal constant liquidationIncentiveMaxMantissa = 1.5e18; // 1.5

    /// @notice The utilization rate balance point which can speeds equals from lenders and borrowers
    uint internal constant balanceUtiRate= 0.5e18 ; // 0.5

    constructor() public {
        admin = msg.sender;
    }

    /*** Assets You Are In ***/

    /**
     * @notice Returns the assets an account has entered
     * @param account The address of the account to pull assets for
     * @return A dynamic list with the assets the account has entered
     */
    function getAssetsIn(address account) external view returns (CToken[] memory) {
        CToken[] memory assetsIn = accountAssets[account];

        return assetsIn;
    }
    /**
     * @notice Returns whether the given account is entered in the given asset
     * @param account The address of the account to check
     * @param cToken The cToken to check
     * @return True if the account is in the asset, otherwise false.
     */
    function checkMembership(address account, CToken cToken) external view returns (bool) {
        return markets[address(cToken)].accountMembership[account];
    }

    /**
     * @notice Add assets to be included in account liquidity calculation
     * @param cTokens The list of addresses of the cToken markets to be enabled
     * @return Success indicator for whether each corresponding market was entered
     */
    function enterMarkets(address[] memory cTokens) public returns (uint[] memory) {
        uint len = cTokens.length;

        uint[] memory results = new uint[](len);
        for (uint i = 0; i < len; i++) {
            CToken cToken = CToken(cTokens[i]);

            results[i] = uint(addToMarketInternal(cToken, msg.sender));
        }

        return results;
    }

    /**
     * @notice Add the market to the borrower's "assets in" for liquidity calculations
     * @param cToken The market to enter
     * @param borrower The address of the account to modify
     * @return Success indicator for whether the market was entered
     */
    function addToMarketInternal(CToken cToken, address borrower) internal returns (Error) {
        Market storage marketToJoin = markets[address(cToken)];

        if (!marketToJoin.isListed) {
            // market is not listed, cannot join
            return Error.MARKET_NOT_LISTED;
        }

        if (marketToJoin.accountMembership[borrower] == true) {
            // already joined
            return Error.NO_ERROR;
        }

        if (accountAssets[borrower].length >= maxAssets)    {
            // no space, cannot join
            return Error.TOO_MANY_ASSETS;
        }// knownsec  判断是否超过最大成员数

        // survived the gauntlet, add to list
        // NOTE: we store these somewhat redundantly as a significant optimization
        //    this avoids having to iterate through the list for the most common use cases
        //    that is, only when we need to perform liquidity checks
        //    and not whenever we want to check if an account is in a particular market
        marketToJoin.accountMembership[borrower] = true;
        accountAssets[borrower].push(cToken);

        emit MarketEntered(cToken, borrower);

        return Error.NO_ERROR;
    }// knownsec  添加成员

    /**
     * @notice Removes asset from sender's account liquidity calculation
     * @dev Sender must not have an outstanding borrow balance in the asset,
     *    or be providing necessary collateral for an outstanding borrow.
     * @param cTokenAddress The address of the asset to be removed
     * @return Whether or not the account successfully exited the market
     */
    function exitMarket(address cTokenAddress) external returns (uint) {
        CToken cToken = CToken(cTokenAddress);
        /* Get sender tokensHeld and amountOwed underlying from the cToken */
        (uint oErr, uint tokensHeld, uint amountOwed, ) = cToken.getAccountSnapshot(msg.sender);
        require(oErr == 0, "exitMarket: getAccountSnapshot failed"); // semi-opaque error code

        /* Fail if the sender has a borrow balance */
        if (amountOwed != 0) {
            return                                          fail(Error.NONZERO_BORROW_BALANCE,
FailureInfo.EXIT_MARKET_BALANCE_OWED);
        }

        /* Fail if the sender is not permitted to redeem all of their tokens */
        uint allowed = redeemAllowedInternal(cTokenAddress, msg.sender, tokensHeld);
        if (allowed != 0) {
            return failOpaque(Error.REJECTION, FailureInfo.EXIT_MARKET_REJECTION, allowed);
        }

        Market storage marketToExit = markets[address(cToken)];

        /* Return true if the sender is not already ‘in’ the market */
        if (!marketToExit.accountMembership[msg.sender]) {
            return uint(Error.NO_ERROR);
        }

        /* Set cToken account membership to false */
        delete marketToExit.accountMembership[msg.sender];

        /* Delete cToken from the account’s list of assets */
        // load into memory for faster iteration
        CToken[] memory userAssetList = accountAssets[msg.sender];
        uint len = userAssetList.length;
        uint assetIndex = len;
        for (uint i = 0; i < len; i++) {
            if (userAssetList[i] == cToken) {
                assetIndex = i;
                break;
            }
        }

        // We *must* have found the asset in the list or our redundant data structure is broken
        assert(assetIndex < len);

        // copy last item in list to location of item to be removed, reduce length by 1
        CToken[] storage storedList = accountAssets[msg.sender];
        storedList[assetIndex] = storedList[storedList.length - 1];
        storedList.length--;

        emit MarketExited(cToken, msg.sender);

        return uint(Error.NO_ERROR);
    }// knownsec 退出

    /*** Policy Hooks ***/

    /**
     * @notice Checks if the account should be allowed to mint tokens in the given market
     * @param cToken The market to verify the mint against
     * @param minter The account which would get the minted tokens
     * @param mintAmount The amount of underlying being supplied to the market in exchange for tokens
     * @return 0 if the mint is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function mintAllowed(address cToken, address minter, uint mintAmount) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!mintGuardianPaused[cToken], "mint is paused");

        // Shh - currently unused
        minter;
        mintAmount;

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // Keep the flywheel moving
        updateCanSupplyIndex(cToken);
        distributeSupplierCan(cToken, minter, false);

        return uint(Error.NO_ERROR);
    }// knownsec 铸币权

    /**
     * @notice Validates mint and reverts on rejection. May emit logs.
     * @param cToken Asset being minted
     * @param minter The address minting the tokens
     * @param actualMintAmount The amount of the underlying asset being minted
     * @param mintTokens The number of tokens being minted
     */
    function mintVerify(address cToken, address minter, uint actualMintAmount, uint mintTokens) external {
        // Shh - currently unused
        cToken;
        minter;
        actualMintAmount;
        mintTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }
    /**
     * @notice Checks if the account should be allowed to redeem tokens in the given market
     * @param cToken The market to verify the redeem against
     * @param redeemer The account which would redeem the tokens
     * @param redeemTokens The number of cTokens to exchange for the underlying asset in the market
     * @return 0 if the redeem is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function redeemAllowed(address cToken, address redeemer, uint redeemTokens) external returns (uint) {
        uint allowed = redeemAllowedInternal(cToken, redeemer, redeemTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }

        // Keep the flywheel moving
        updateCanSupplyIndex(cToken);
        distributeSupplierCan(cToken, redeemer, false);

        return uint(Error.NO_ERROR);
    }// knownsec 外部赎回权限
    function redeemAllowedInternal(address cToken, address redeemer, uint redeemTokens) internal view returns
(uint) {
        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        /* If the redeemer is not 'in' the market, then we can bypass the liquidity check */
        if (!markets[cToken].accountMembership[redeemer]) {
            return uint(Error.NO_ERROR);
        }

        /* Otherwise, perform a hypothetical liquidity check to guard against shortfall */
        (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(redeemer, CToken(cToken),
redeemTokens, 0);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall > 0) {
            return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        return uint(Error.NO_ERROR);
    }// knownsec 内部赎回权限

    /**
     * @notice Validates redeem and reverts on rejection. May emit logs.
     * @param cToken Asset being redeemed
     * @param redeemer The address redeeming the tokens
     * @param redeemAmount The amount of the underlying asset being redeemed
     * @param redeemTokens The number of tokens being redeemed
     */
    function redeemVerify(address cToken, address redeemer, uint redeemAmount, uint redeemTokens) external {
        // Shh - currently unused
        cToken;
        redeemer;

        // Require tokens is zero or amount is also zero
        if (redeemTokens == 0 && redeemAmount > 0) {
            revert("redeemTokens zero");
        }
    }

    /**
     * @notice Checks if the account should be allowed to borrow the underlying asset of the given market
     * @param cToken The market to verify the borrow against
     * @param borrower The account which would borrow the asset
     * @param borrowAmount The amount of underlying the account would borrow
     * @return 0 if the borrow is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function borrowAllowed(address cToken, address borrower, uint borrowAmount) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!borrowGuardianPaused[cToken], "borrow is paused");

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (!markets[cToken].accountMembership[borrower]) {
            // only cTokens may call borrowAllowed if borrower not in market
            require(msg.sender == cToken, "sender must be cToken");

            // attempt to add borrower to the market
            Error err = addToMarketInternal(CToken(msg.sender), borrower);
            if (err != Error.NO_ERROR) {
                return uint(err);
            }

            // it should be impossible to break the important invariant
            assert(markets[cToken].accountMembership[borrower]);
        }

        if (oracle.getUnderlyingPrice(CToken(cToken)) == 0) {
            return uint(Error.PRICE_ERROR);
        }

        (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(borrower, CToken(cToken), 0,
borrowAmount);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall > 0) {
            return uint(Error.INSUFFICIENT_LIQUIDITY);
        }

        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
        updateCanBorrowIndex(cToken, borrowIndex);
        distributeBorrowerCan(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
    }// knownsec 借贷权限

    /**
     * @notice Validates borrow and reverts on rejection. May emit logs.
     * @param cToken Asset whose underlying is being borrowed
     * @param borrower The address borrowing the underlying
     * @param borrowAmount The amount of the underlying asset requested to borrow
     */
    function borrowVerify(address cToken, address borrower, uint borrowAmount) external {
        // Shh - currently unused
        cToken;
        borrower;
        borrowAmount;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to repay a borrow in the given market
     * @param cToken The market to verify the repay against
     * @param payer The account which would repay the asset
     * @param borrower The account which would borrowed the asset
     * @param repayAmount The amount of the underlying asset the account would repay
     * @return 0 if the repay is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function repayBorrowAllowed(
        address cToken,
        address payer,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        payer;
        borrower;
        repayAmount;

        if (!markets[cToken].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        // Keep the flywheel moving
        Exp memory borrowIndex = Exp({mantissa: CToken(cToken).borrowIndex()});
        updateCanBorrowIndex(cToken, borrowIndex);
        distributeBorrowerCan(cToken, borrower, borrowIndex, false);

        return uint(Error.NO_ERROR);
    }// knownsec 偿还权限

    /**
     * @notice Validates repayBorrow and reverts on rejection. May emit logs.
     * @param cToken Asset being repaid
     * @param payer The address repaying the borrow
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function repayBorrowVerify(
        address cToken,
        address payer,
        address borrower,
        uint actualRepayAmount,
        uint borrowerIndex) external {
        // Shh - currently unused
        cToken;
        payer;
        borrower;
        actualRepayAmount;
        borrowerIndex;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the liquidation should be allowed to occur
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param repayAmount The amount of underlying being repaid
     */
    function liquidateBorrowAllowed(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint repayAmount) external returns (uint) {
        // Shh - currently unused
        liquidator;

        if (!markets[cTokenBorrowed].isListed || !markets[cTokenCollateral].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        /* The borrower must have shortfall in order to be liquidatable */
        (Error err, , uint shortfall) = getAccountLiquidityInternal(borrower);
        if (err != Error.NO_ERROR) {
            return uint(err);
        }
        if (shortfall == 0) {
            return uint(Error.INSUFFICIENT_SHORTFALL);
        }

        /* The liquidator may not repay more than what is allowed by the closeFactor */
        uint borrowBalance = CToken(cTokenBorrowed).borrowBalanceStored(borrower);
        (MathError mathErr, uint maxClose) = mulScalarTruncate(Exp({mantissa: closeFactorMantissa}),
borrowBalance);
        if (mathErr != MathError.NO_ERROR) {
            return uint(Error.MATH_ERROR);
        }
        if (repayAmount > maxClose) {
            return uint(Error.TOO_MUCH_REPAY);
        }

        return uint(Error.NO_ERROR);
    }// knownsec 结算权限

    /**
     * @notice Validates liquidateBorrow and reverts on rejection. May emit logs.
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param actualRepayAmount The amount of underlying being repaid
     */
    function liquidateBorrowVerify(
        address cTokenBorrowed,
        address cTokenCollateral,
        address liquidator,
        address borrower,
        uint actualRepayAmount,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenBorrowed;
        cTokenCollateral;
        liquidator;
        borrower;
        actualRepayAmount;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the seizing of assets should be allowed to occur
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeAllowed(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external returns (uint) {
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!seizeGuardianPaused, "seize is paused");

        // Shh - currently unused
        seizeTokens;

        if (!markets[cTokenCollateral].isListed || !markets[cTokenBorrowed].isListed) {
            return uint(Error.MARKET_NOT_LISTED);
        }

        if (CToken(cTokenCollateral).comptroller() != CToken(cTokenBorrowed).comptroller()) {
            return uint(Error.ChannelsTROLLER_MISMATCH);
        }

        // Keep the flywheel moving
        updateCanSupplyIndex(cTokenCollateral);
        distributeSupplierCan(cTokenCollateral, borrower, false);
        distributeSupplierCan(cTokenCollateral, liquidator, false);

        return uint(Error.NO_ERROR);
    }// knownsec  抵押权限

    /**
     * @notice Validates seize and reverts on rejection. May emit logs.
     * @param cTokenCollateral Asset which was used as collateral and will be seized
     * @param cTokenBorrowed Asset which was borrowed by the borrower
     * @param liquidator The address repaying the borrow and seizing the collateral
     * @param borrower The address of the borrower
     * @param seizeTokens The number of collateral tokens to seize
     */
    function seizeVerify(
        address cTokenCollateral,
        address cTokenBorrowed,
        address liquidator,
        address borrower,
        uint seizeTokens) external {
        // Shh - currently unused
        cTokenCollateral;
        cTokenBorrowed;
        liquidator;
        borrower;
        seizeTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /**
     * @notice Checks if the account should be allowed to transfer tokens in the given market
     * @param cToken The market to verify the transfer against
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     * @return 0 if the transfer is allowed, otherwise a semi-opaque error code (See ErrorReporter.sol)
     */
    function transferAllowed(address cToken, address src, address dst, uint transferTokens) external returns (uint)
{
        // Pausing is a very serious situation - we revert to sound the alarms
        require(!transferGuardianPaused, "transfer is paused");

        // Currently the only consideration is whether or not
        //   the src is allowed to redeem this many tokens
        uint allowed = redeemAllowedInternal(cToken, src, transferTokens);
        if (allowed != uint(Error.NO_ERROR)) {
            return allowed;
        }

        // Keep the flywheel moving
        updateCanSupplyIndex(cToken);
        distributeSupplierCan(cToken, src, false);
        distributeSupplierCan(cToken, dst, false);

        return uint(Error.NO_ERROR);
    }// knownsec  转账权限
    /**
     * @notice Validates transfer and reverts on rejection. May emit logs.
     * @param cToken Asset being transferred
     * @param src The account which sources the tokens
     * @param dst The account which receives the tokens
     * @param transferTokens The number of cTokens to transfer
     */
    function transferVerify(address cToken, address src, address dst, uint transferTokens) external {
        // Shh - currently unused
        cToken;
        src;
        dst;
        transferTokens;

        // Shh - we don't ever want this hook to be marked pure
        if (false) {
            maxAssets = maxAssets;
        }
    }

    /*** Liquidity/Liquidation Calculations ***/

    /**
     * @dev Local vars for avoiding stack-depth limits in calculating account liquidity.
     *  Note that `cTokenBalance` is the number of cTokens the account owns in the market,
     *  whereas `borrowBalance` is the amount of underlying that the account has borrowed.
     */
    struct AccountLiquidityLocalVars {
        uint sumCollateral;
        uint sumBorrowPlusEffects;
        uint cTokenBalance;
        uint borrowBalance;
        uint exchangeRateMantissa;
        uint oraclePriceMantissa;
        Exp collateralFactor;
        Exp exchangeRate;
        Exp oraclePrice;
        Exp tokensToDenom;
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code (semi-opaque),
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidity(address account) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account, CToken(0),
0, 0);

        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine the current account liquidity wrt collateral requirements
     * @return (possible error code,
                account liquidity in excess of collateral requirements,
     *          account shortfall below collateral requirements)
     */
    function getAccountLiquidityInternal(address account) internal view returns (Error, uint, uint) {
        return getHypotheticalAccountLiquidityInternal(account, CToken(0), 0, 0);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @return (possible error code (semi-opaque),
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidity(
        address account,
        address cTokenModify,
        uint redeemTokens,
        uint borrowAmount) public view returns (uint, uint, uint) {
        (Error err, uint liquidity, uint shortfall) = getHypotheticalAccountLiquidityInternal(account,
CToken(cTokenModify), redeemTokens, borrowAmount);
        return (uint(err), liquidity, shortfall);
    }

    /**
     * @notice Determine what the account liquidity would be if the given amounts were redeemed/borrowed
     * @param cTokenModify The market to hypothetically redeem/borrow in
     * @param account The account to determine liquidity for
     * @param redeemTokens The number of tokens to hypothetically redeem
     * @param borrowAmount The amount of underlying to hypothetically borrow
     * @dev Note that we calculate the exchangeRateStored for each collateral cToken using stored data,
     *  without calculating accumulated interest.
     * @return (possible error code,
                hypothetical account liquidity in excess of collateral requirements,
     *          hypothetical account shortfall below collateral requirements)
     */
    function getHypotheticalAccountLiquidityInternal(
        address account,
        CToken cTokenModify,
        uint redeemTokens,
        uint borrowAmount) internal view returns (Error, uint, uint) {

        AccountLiquidityLocalVars memory vars; // Holds all our calculation results
        uint oErr;
        MathError mErr;

        // For each asset the account is in
        CToken[] memory assets = accountAssets[account];
        for (uint i = 0; i < assets.length; i++) {
            CToken asset = assets[i];

            // Read the balances and exchange rate from the cToken
            (oErr, vars.cTokenBalance, vars.borrowBalance, vars.exchangeRateMantissa) =
asset.getAccountSnapshot(account);
            if (oErr != 0) { // semi-opaque error code, we assume NO_ERROR == 0 is invariant between
upgrades
                return (Error.SNAPSHOT_ERROR, 0, 0);
            }
            vars.collateralFactor = Exp({mantissa: markets[address(asset)].collateralFactorMantissa});
            vars.exchangeRate = Exp({mantissa: vars.exchangeRateMantissa});

            // Get the normalized price of the asset
            vars.oraclePriceMantissa = oracle.getUnderlyingPrice(asset);
            if (vars.oraclePriceMantissa == 0) {
                return (Error.PRICE_ERROR, 0, 0);
            }
            vars.oraclePrice = Exp({mantissa: vars.oraclePriceMantissa});

            // Pre-canute a conversion factor from tokens -> ht (normalized price value)
            (mErr, vars.tokensToDenom) = mulExp3(vars.collateralFactor, vars.exchangeRate,
vars.oraclePrice);
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // sumCollateral += tokensToDenom * cTokenBalance
            (mErr, vars.sumCollateral) = mulScalarTruncateAddUInt(vars.tokensToDenom,
vars.cTokenBalance, vars.sumCollateral);
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // sumBorrowPlusEffects += oraclePrice * borrowBalance
            (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice,
vars.borrowBalance, vars.sumBorrowPlusEffects);
            if (mErr != MathError.NO_ERROR) {
                return (Error.MATH_ERROR, 0, 0);
            }

            // Calculate effects of interacting with cTokenModify
            if (asset == cTokenModify) {
                // redeem effect
                // sumBorrowPlusEffects += tokensToDenom * redeemTokens
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.tokensToDenom,
redeemTokens, vars.sumBorrowPlusEffects);
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }

                // borrow effect
                // sumBorrowPlusEffects += oraclePrice * borrowAmount
                (mErr, vars.sumBorrowPlusEffects) = mulScalarTruncateAddUInt(vars.oraclePrice,
borrowAmount, vars.sumBorrowPlusEffects);
                if (mErr != MathError.NO_ERROR) {
                    return (Error.MATH_ERROR, 0, 0);
                }
            }
        }

        // These are safe, as the underflow condition is checked first
        if (vars.sumCollateral > vars.sumBorrowPlusEffects) {
            return (Error.NO_ERROR, vars.sumCollateral - vars.sumBorrowPlusEffects, 0);
        } else {
            return (Error.NO_ERROR, 0, vars.sumBorrowPlusEffects - vars.sumCollateral);
        }
    }// knownsec  获取流动性变化

    /**
     * @notice Calculate number of tokens of collateral asset to seize given an underlying amount
     * @dev Used in liquidation (called in cToken.liquidateBorrowFresh)
     * @param cTokenBorrowed The address of the borrowed cToken
     * @param cTokenCollateral The address of the collateral cToken
     * @param actualRepayAmount The amount of cTokenBorrowed underlying to convert into cTokenCollateral
tokens
     * @return (errorCode, number of cTokenCollateral tokens to be seized in a liquidation)
     */
    function liquidateCalculateSeizeTokens(address cTokenBorrowed, address cTokenCollateral, uint
actualRepayAmount) external view returns (uint, uint) {
        /* Read oracle prices for borrowed and collateral markets */
        uint priceBorrowedMantissa = oracle.getUnderlyingPrice(CToken(cTokenBorrowed));
        uint priceCollateralMantissa = oracle.getUnderlyingPrice(CToken(cTokenCollateral));
        if (priceBorrowedMantissa == 0 || priceCollateralMantissa == 0) {
            return (uint(Error.PRICE_ERROR), 0);
        }

        /*
         * Get the exchange rate and calculate the number of collateral tokens to seize:
         *   seizeAmount = actualRepayAmount * liquidationIncentive * priceBorrowed / priceCollateral
         *   seizeTokens = seizeAmount / exchangeRate
         *     = actualRepayAmount * (liquidationIncentive * priceBorrowed) / (priceCollateral * exchangeRate)
         */
        uint exchangeRateMantissa = CToken(cTokenCollateral).exchangeRateStored(); // Note: reverts on error
        uint seizeTokens;
        Exp memory numerator;
        Exp memory denominator;
        Exp memory ratio;
        MathError mathErr;

        (mathErr, numerator) = mulExp(liquidationIncentiveMantissa, priceBorrowedMantissa);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, denominator) = mulExp(priceCollateralMantissa, exchangeRateMantissa);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, ratio) = divExp(numerator, denominator);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        (mathErr, seizeTokens) = mulScalarTruncate(ratio, actualRepayAmount);
        if (mathErr != MathError.NO_ERROR) {
            return (uint(Error.MATH_ERROR), 0);
        }

        return (uint(Error.NO_ERROR), seizeTokens);
    }// knownesc  获取抵押物价值

    /*** Admin Functions ***/

    /**
      * @notice Sets a new price oracle for the comptroller
      * @dev Admin function to set a new price oracle
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPriceOracle(PriceOracle newOracle) public returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PRICE_ORACLE_OWNER_CHECK);
        }

        // Track the old oracle for the comptroller
        PriceOracle oldOracle = oracle;

        // Set comptroller's oracle to newOracle
        oracle = newOracle;

        // Emit NewPriceOracle(oldOracle, newOracle)
        emit NewPriceOracle(oldOracle, newOracle);

        return uint(Error.NO_ERROR);
    }// knownsec  设置新的预言机

    /**
      * @notice Sets the closeFactor used when liquidating borrows
      * @dev Admin function to set closeFactor
      * @param newCloseFactorMantissa New close factor, scaled by 1e18
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
    */
function _setCloseFactor(uint newCloseFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_CLOSE_FACTOR_OWNER_CHECK);
        }

        Exp memory newCloseFactorExp = Exp({mantissa: newCloseFactorMantissa});
        Exp memory lowLimit = Exp({mantissa: closeFactorMinMantissa});
        if (lessThanOrEqualExp(newCloseFactorExp, lowLimit)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        Exp memory highLimit = Exp({mantissa: closeFactorMaxMantissa});
        if (lessThanExp(highLimit, newCloseFactorExp)) {
            return fail(Error.INVALID_CLOSE_FACTOR, FailureInfo.SET_CLOSE_FACTOR_VALIDATION);
        }

        uint oldCloseFactorMantissa = closeFactorMantissa;
        closeFactorMantissa = newCloseFactorMantissa;
        emit NewCloseFactor(oldCloseFactorMantissa, closeFactorMantissa);

        return uint(Error.NO_ERROR);
}// knownsec  设置应结算元素

/**
    * @notice Sets the collateralFactor for a market
    * @dev Admin function to set per-market collateralFactor
    * @param cToken The market to set the factor on
    * @param newCollateralFactorMantissa The new collateral factor, scaled by 1e18
    * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
    */
function _setCollateralFactor(CToken cToken, uint newCollateralFactorMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return                                                  fail(Error.UNAUTHORIZED,
FailureInfo.SET_COLLATERAL_FACTOR_OWNER_CHECK);
        }

        // Verify market is listed
        Market storage market = markets[address(cToken)];
        if (!market.isListed) {
            return                                             fail(Error.MARKET_NOT_LISTED,
FailureInfo.SET_COLLATERAL_FACTOR_NO_EXISTS);
        }

        Exp memory newCollateralFactorExp = Exp({mantissa: newCollateralFactorMantissa});

        // Check collateral factor <= 0.9
        Exp memory highLimit = Exp({mantissa: collateralFactorMaxMantissa});
        if (lessThanExp(highLimit, newCollateralFactorExp)) {
            return                                    fail(Error.INVALID_COLLATERAL_FACTOR,
FailureInfo.SET_COLLATERAL_FACTOR_VALIDATION);
        }

        // If collateral factor != 0, fail if price == 0
        if (newCollateralFactorMantissa != 0 && oracle.getUnderlyingPrice(cToken) == 0) {
            return fail(Error.PRICE_ERROR, FailureInfo.SET_COLLATERAL_FACTOR_WITHOUT_PRICE);
        }

        // Set market's collateral factor to new collateral factor, remember old value
        uint oldCollateralFactorMantissa = market.collateralFactorMantissa;
        market.collateralFactorMantissa = newCollateralFactorMantissa;

        // Emit event with asset, old collateral factor, and new collateral factor
        emit NewCollateralFactor(cToken, oldCollateralFactorMantissa, newCollateralFactorMantissa);

        return uint(Error.NO_ERROR);
}// knownsec  设置可抵押元素

/**
    * @notice Sets maxAssets which controls how many markets can be entered
    * @dev Admin function to set maxAssets
    * @param newMaxAssets New max assets
    * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
    */
function _setMaxAssets(uint newMaxAssets) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_MAX_ASSETS_OWNER_CHECK);
        }

        uint oldMaxAssets = maxAssets;
        maxAssets = newMaxAssets;
        emit NewMaxAssets(oldMaxAssets, newMaxAssets);

        return uint(Error.NO_ERROR);
    }// knownsec  设置最大成员数

    /**
      * @notice Sets liquidationIncentive
      * @dev Admin function to set liquidationIncentive
      * @param newLiquidationIncentiveMantissa New liquidationIncentive scaled by 1e18
      * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
      */
    function _setLiquidationIncentive(uint newLiquidationIncentiveMantissa) external returns (uint) {
        // Check caller is admin
        if (msg.sender != admin) {
            return                                          fail(Error.UNAUTHORIZED,
FailureInfo.SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
        }

        // Check de-scaled min <= newLiquidationIncentive <= max
        Exp memory newLiquidationIncentive = Exp({mantissa: newLiquidationIncentiveMantissa});
        Exp memory minLiquidationIncentive = Exp({mantissa: liquidationIncentiveMinMantissa});
        if (lessThanExp(newLiquidationIncentive, minLiquidationIncentive)) {
            return                              fail(Error.INVALID_LIQUIDATION_INCENTIVE,
FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
        }

        Exp memory maxLiquidationIncentive = Exp({mantissa: liquidationIncentiveMaxMantissa});
        if (lessThanExp(maxLiquidationIncentive, newLiquidationIncentive)) {
            return                              fail(Error.INVALID_LIQUIDATION_INCENTIVE,
FailureInfo.SET_LIQUIDATION_INCENTIVE_VALIDATION);
        }

        // Save current value for use in log
        uint oldLiquidationIncentiveMantissa = liquidationIncentiveMantissa;

        // Set liquidation incentive to new incentive
        liquidationIncentiveMantissa = newLiquidationIncentiveMantissa;

        // Emit event with old incentive, new incentive
        emit NewLiquidationIncentive(oldLiquidationIncentiveMantissa, newLiquidationIncentiveMantissa);

        return uint(Error.NO_ERROR);
    }// knownsec  设置结算奖励

    /**
      * @notice Add the market to the markets mapping and set it as listed
      * @dev Admin function to set isListed and add support for the market
      * @param cToken The address of the market (token) to list
      * @return uint 0=success, otherwise a failure. (See enum Error for details)
      */
    function _supportMarket(CToken cToken) external returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SUPPORT_MARKET_OWNER_CHECK);
        }

        if (markets[address(cToken)].isListed) {
            return fail(Error.MARKET_ALREADY_LISTED, FailureInfo.SUPPORT_MARKET_EXISTS);
        }

        cToken.isCToken(); // Sanity check to make sure its really a CToken

        markets[address(cToken)] = Market({isListed: true, isCaned: false, collateralFactorMantissa: 0});

        _addMarketInternal(address(cToken));

        emit MarketListed(cToken);

        return uint(Error.NO_ERROR);
    }// knownsec  市场清单

    function _addMarketInternal(address cToken) internal {
        for (uint i = 0; i < allMarkets.length; i ++) {
            require(allMarkets[i] != CToken(cToken), "market already added");
        }
        allMarkets.push(CToken(cToken));
    }// knownsec  添加新市场

    /**
     * @notice Admin function to change the Pause Guardian
     * @param newPauseGuardian The address of the new Pause Guardian
     * @return uint 0=success, otherwise a failure. (See enum Error for details)
     */
    function _setPauseGuardian(address newPauseGuardian) public returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PAUSE_GUARDIAN_OWNER_CHECK);
        }

        // Save current value for inclusion in log
        address oldPauseGuardian = pauseGuardian;
        // Store pauseGuardian with value newPauseGuardian
        pauseGuardian = newPauseGuardian;

        // Emit NewPauseGuardian(OldPauseGuardian, NewPauseGuardian)
        emit NewPauseGuardian(oldPauseGuardian, pauseGuardian);

        return uint(Error.NO_ERROR);
    }// knownsec  设置监管人员

    function _setMintPaused(CToken cToken, bool state) public returns (bool) {
        require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can
pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        mintGuardianPaused[address(cToken)] = state;
        emit ActionPaused(cToken, "Mint", state);
        return state;
    }// knownsec  控制铸币

    function _setBorrowPaused(CToken cToken, bool state) public returns (bool) {
        require(markets[address(cToken)].isListed, "cannot pause a market that is not listed");
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can
pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        borrowGuardianPaused[address(cToken)] = state;
        emit ActionPaused(cToken, "Borrow", state);
        return state;
    }// knownsec  控制借贷

    function _setTransferPaused(bool state) public returns (bool) {
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can
pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        transferGuardianPaused = state;
        emit ActionPaused("Transfer", state);
        return state;
    }// knownsec  控制转账

    function _setSeizePaused(bool state) public returns (bool) {
        require(msg.sender == pauseGuardian || msg.sender == admin, "only pause guardian and admin can
pause");
        require(msg.sender == admin || state == true, "only admin can unpause");

        seizeGuardianPaused = state;
        emit ActionPaused("Seize", state);
        return state;
    }// knownsec  控制抵押

    function _become(Unitroller unitroller) public {
        require(msg.sender == unitroller.admin(), "only unitroller admin can change brains");
        require(unitroller._acceptImplementation() == 0, "change not authorized");
    }


    /**
      * @notice Checks caller is admin, or this contract is becoming the new implementation
      */
    function adminOrInitializing() internal view returns (bool) {
        return msg.sender == admin || msg.sender == comptrollerImplementation;
    }

    /*** Can Distribution ***/

    /**
      * @notice Accrue Channels to the market by updating the supply index
      * @param cToken The market whose supply index to update
      */
    function updateCanSupplyIndex(address cToken) internal {
        CanMarketState storage supplyState = canSupplyState[cToken];
        uint supplySpeed = getSupplySpeed(cToken);
        uint blockNumber = getBlockNumber();
        uint deltaBlocks = sub_(blockNumber, uint(supplyState.block));
        if (deltaBlocks > 0 && supplySpeed > 0) {
            uint supplyTokens = CToken(cToken).totalSupply();
            uint canAccrued = mul_(deltaBlocks, supplySpeed);
            Double memory ratio = supplyTokens > 0 ? fraction(canAccrued, supplyTokens) : Double({mantissa:
0});
            Double memory index = add_(Double({mantissa: supplyState.index}), ratio);
            canSupplyState[cToken] = CanMarketState({
            index: safe224(index.mantissa, "new index exceeds 224 bits"),
            block: safe32(blockNumber, "block number exceeds 32 bits")
            });
        } else if (deltaBlocks > 0) {
            supplyState.block = safe32(blockNumber, "block number exceeds 32 bits");
        }
    }// knownsec  更新 can 代币发行指数

    /**
      * @notice Accrue Channels to the market by updating the borrow index
      * @param cToken The market whose borrow index to update
      */
    function updateCanBorrowIndex(address cToken, Exp memory marketBorrowIndex) internal {
        CanMarketState storage borrowState = canBorrowState[cToken];
        uint borrowSpeed = getBorrowSpeed(cToken);
        uint blockNumber = getBlockNumber();
        uint deltaBlocks = sub_(blockNumber, uint(borrowState.block));
        if (deltaBlocks > 0 && borrowSpeed > 0) {
            uint borrowAmount = div_(CToken(cToken).totalBorrows(), marketBorrowIndex);
            uint canAccrued = mul_(deltaBlocks, borrowSpeed);
            Double memory ratio = borrowAmount > 0 ? fraction(canAccrued, borrowAmount) :
Double({mantissa: 0});
            Double memory index = add_(Double({mantissa: borrowState.index}), ratio);
            canBorrowState[cToken] = CanMarketState({
            index: safe224(index.mantissa, "new index exceeds 224 bits"),
            block: safe32(blockNumber, "block number exceeds 32 bits")
            });
        } else if (deltaBlocks > 0) {
            borrowState.block = safe32(blockNumber, "block number exceeds 32 bits");
        }
    }// knownsec  更新 can 代币借贷指数

    function getSupplySpeed(address cToken) public view returns (uint){

        uint utilizationRate = getUtilizationRate(cToken);

        uint supplySpeed;
        if (utilizationRate == 0){
            supplySpeed = canSpeeds[cToken];
            return supplySpeed;
        }
        if (utilizationRate <= balanceUtiRate){
            uint leftPercentage = utilizationRate.mul(5e17).div(balanceUtiRate);
            supplySpeed = canSpeeds[cToken].mul(leftPercentage).div(1e18);
        }else {
            uint slop = (5e35).div((1e18).sub(balanceUtiRate));
            uint rightPercentage = (5e17).add(slop.mul(utilizationRate.sub(balanceUtiRate)).div(1e18));
            supplySpeed = canSpeeds[cToken].mul(rightPercentage).div(1e18);
        }
        return supplySpeed;
    }// knownsec  获取发行速度

    function getBorrowSpeed(address cToken) public view returns (uint){
        uint supplySpeed = getSupplySpeed(cToken);
        uint borrowSpeed = canSpeeds[cToken] > supplySpeed ? canSpeeds[cToken].sub(supplySpeed) : 0;
        return borrowSpeed;
    }// knownsec  获取借贷速度

    /**
      * @notice Calculate Channels accrued by a supplier and possibly transfer it to them
      * @param cToken The market in which the supplier is interacting
      * @param supplier The address of the supplier to distribute Channels to
      */
    function distributeSupplierCan(address cToken, address supplier, bool distributeAll) internal {
        CanMarketState storage supplyState = canSupplyState[cToken];
        Double memory supplyIndex = Double({mantissa: supplyState.index});
        Double memory supplierIndex = Double({mantissa: canSupplierIndex[cToken][supplier]});
        canSupplierIndex[cToken][supplier] = supplyIndex.mantissa;

        if (supplierIndex.mantissa == 0 && supplyIndex.mantissa > 0) {
            supplierIndex.mantissa = canInitialIndex;
        }

        Double memory deltaIndex = sub_(supplyIndex, supplierIndex);
        uint supplierTokens = CToken(cToken).balanceOf(supplier);
        uint supplierDelta = mul_(supplierTokens, deltaIndex);
        uint supplierAccrued = add_(canAccrued[supplier], supplierDelta);
        canAccrued[supplier] = transferCan(supplier, supplierAccrued, distributeAll ? 0 : canClaimThreshold);
        emit DistributedSupplierCan(CToken(cToken), supplier, supplierDelta, supplyIndex.mantissa);
    }// knownsec  分配 Can 代币

    /**
      * @notice Calculate Channels accrued by a borrower and possibly transfer it to them
      * @dev Borrowers will not begin to accrue until after the first interaction with the protocol.
      * @param cToken The market in which the borrower is interacting
      * @param borrower The address of the borrower to distribute Channels to
      */
    function distributeBorrowerCan(address cToken, address borrower, Exp memory marketBorrowIndex, bool
distributeAll) internal {
        CanMarketState storage borrowState = canBorrowState[cToken];
        Double memory borrowIndex = Double({mantissa: borrowState.index});
        Double memory borrowerIndex = Double({mantissa: canBorrowerIndex[cToken][borrower]});
        canBorrowerIndex[cToken][borrower] = borrowIndex.mantissa;

        if (borrowerIndex.mantissa > 0) {
            Double memory deltaIndex = sub_(borrowIndex, borrowerIndex);
            uint borrowerAmount = div_(CToken(cToken).borrowBalanceStored(borrower), marketBorrowIndex);
            uint borrowerDelta = mul_(borrowerAmount, deltaIndex);
            uint borrowerAccrued = add_(canAccrued[borrower], borrowerDelta);
            canAccrued[borrower] = transferCan(borrower, borrowerAccrued, distributeAll ? 0 : canClaimThreshold);
            emit DistributedBorrowerCan(CToken(cToken), borrower, borrowerDelta, borrowIndex.mantissa);
        }
    }// knownsec 分配借款人 Can 代币

    /**
      * @notice Transfer Channels to the user, if they are above the threshold
      * @dev Note: If there is not enough Channels, we do not perform the transfer all.
      * @param user The address of the user to transfer Channels to
      * @param userAccrued The amount of Channels to (possibly) transfer
      * @return The amount of Channels which was NOT transferred to the user
      */
    function transferCan(address user, uint userAccrued, uint threshold) internal returns (uint) {
        if (userAccrued >= threshold && userAccrued > 0) {
            Can can = Can(getCanAddress());
            uint canRemaining = can.balanceOf(address(this));
            if (userAccrued <= canRemaining) {
                can.transfer(user, userAccrued);
                return 0;
            }
        }
        return userAccrued;
    }// knownsec 转账 can 代币功能

    /**
      * @notice Claim all the can accrued by holder in all markets
      * @param holder The address to claim Channels for
      */
    function claimCan(address holder) public {
        return claimCan(holder, allMarkets);
    }

    /**
      * @notice Claim all the can accrued by holder in the specified markets
      * @param holder The address to claim Channels for
      * @param cTokens The list of markets to claim Channels in
      */
    function claimCan(address holder, CToken[] memory cTokens) public {
        address[] memory holders = new address[](1);
        holders[0] = holder;
        claimCan(holders, cTokens, true, true);
    }

    /**
      * @notice Claim all can accrued by the holders
      * @param holders The addresses to claim Channels for
      * @param cTokens The list of markets to claim Channels in
      * @param borrowers Whether or not to claim Channels earned by borrowing
      * @param suppliers Whether or not to claim Channels earned by supplying
      */
    function claimCan(address[] memory holders, CToken[] memory cTokens, bool borrowers, bool suppliers) public {
        for (uint i = 0; i < cTokens.length; i++) {
            CToken cToken = cTokens[i];
            require(markets[address(cToken)].isListed, "market must be listed");
            if (borrowers == true) {
                Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
                updateCanBorrowIndex(address(cToken), borrowIndex);
                for (uint j = 0; j < holders.length; j++) {
                    distributeBorrowerCan(address(cToken), holders[j], borrowIndex, true);
                }
            }
            if (suppliers == true) {
                updateCanSupplyIndex(address(cToken));
                for (uint j = 0; j < holders.length; j++) {
                    distributeSupplierCan(address(cToken), holders[j], true);
                }
            }
        }
    }// knownsec 累计诉求

    /*** Can Distribution Admin ***/

    /**
      * @notice Set cTokens canSpeed
      * @param cTokens The addresses of cTokens
      * @param speeds The list of CAN speeds
```

```
    */
    function _setCanSpeeds(address[] memory cTokens, uint[] memory speeds) public {
        require(msg.sender == admin, "only admin can set can speeds");

        uint numMarkets = cTokens.length;
        uint numSpeeds = speeds.length;

        require(numMarkets != 0 && numMarkets == numSpeeds, "invalid input");

        for (uint i = 0; i < numMarkets; i++) {
            if (speeds[i] > 0) {
                _initCanState(cTokens[i]);
            }

            CToken cToken = CToken(cTokens[i]);
            Exp memory borrowIndex = Exp({mantissa: cToken.borrowIndex()});
            updateCanSupplyIndex(address(cToken));
            updateCanBorrowIndex(address(cToken), borrowIndex);

            canSpeeds[address(cToken)] = speeds[i];
            emit CanSpeedUpdated(cToken, speeds[i]);
        }
    }

    function _initCanState(address cToken) internal {
        if (canSupplyState[cToken].index == 0 && canSupplyState[cToken].block == 0) {
            canSupplyState[cToken] = CanMarketState({
            index: canInitialIndex,
            block: safe32(getBlockNumber(), "block number exceeds 32 bits")
            });
        }

        if (canBorrowState[cToken].index == 0 && canBorrowState[cToken].block == 0) {
            canBorrowState[cToken] = CanMarketState({
            index: canInitialIndex,
            block: safe32(getBlockNumber(), "block number exceeds 32 bits")
            });
        }
    }



    function getUtilizationRate(address cToken) public view returns (uint) {
        CToken _cToken = CToken(cToken);
        uint _cash = _cToken.getCash();
        uint _borrow = _cToken.totalBorrows();
        uint _reserve = _cToken.totalReserves();

        if (_borrow == 0 || sub_(add_(_cash,_borrow), _reserve) == 0) {
            return 0;
        }

        return div_(mul_(_borrow,1e18),sub_(add_(_cash,_borrow), _reserve));
    }

    /**
     * @notice Return all of the markets
     * @dev The automatic getter may be used to access an individual market.
     * @return The list of market addresses
     */
    function getAllMarkets() public view returns (CToken[] memory) {
        return allMarkets;
    }

    function getBlockNumber() public view returns (uint) {
        return block.number;
    }

    /**
     * @notice Return the address of the Channels token
     * @return The address of Channels
     */
    function getCanAddress() public view returns (address) {
        return 0x1e6395E6B059fc97a4ddA925b6c5ebf19E05c69f;          // todo: update to new can address
    }
}
```

ExchangePool.sol

```
pragma solidity ^0.5.16;

library Math {

    function max(uint256 a, uint256 b) internal pure returns (uint256) {
        return a >= b ? a : b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a < b ? a : b;
    }

    function average(uint256 a, uint256 b) internal pure returns (uint256) {
        return (a / 2) + (b / 2) + ((a % 2 + b % 2) / 2);
    }
}

library SafeMath {

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        if (a == 0) {
            return 0;
        }

        uint256 c = a * b;
        require(c / a == b, "SafeMath: multiplication overflow");

        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        return div(a, b, "SafeMath: division by zero");
    }

    function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b > 0, errorMessage);
        uint256 c = a / b;

        return c;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        return mod(a, b, "SafeMath: modulo by zero");
    }

    function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b != 0, errorMessage);
        return a % b;
    }
}
contract Context {

    constructor () internal { }

    function _msgSender() internal view returns (address payable) {
        return msg.sender;
    }

    function _msgData() internal view returns (bytes memory) {
        this;
        return msg.data;
    }
}

contract Ownable is Context {
    address private _owner;
    address private _pendingOwner;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);
    event OwnershipAccepted(address indexed pendingOwner);

    constructor () internal {
        _owner = _msgSender();
        emit OwnershipTransferred(address(0), _owner);
    }

    function owner() public view returns (address) {
        return _owner;
    }

    modifier onlyOwner() {
        require(isOwner(), "Ownable: caller is not the owner");
        _;
    }

    function isOwner() public view returns (bool) {
        return _msgSender() == _owner;
    }

    function renounceOwnership() public onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

    function transferOwnership(address newOwner) public onlyOwner {
        _transferOwnership(newOwner);
    }

    function _transferOwnership(address newPendingOwner) internal {
        require(newPendingOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newPendingOwner);
        _pendingOwner = newPendingOwner;
    }

    function acceptOwnership() public{
        require(_msgSender() == _pendingOwner, "Ownable: caller is not pending owner");
        emit OwnershipAccepted(_pendingOwner);
        _owner = _pendingOwner;
    }
}

interface IERC20 {

    function totalSupply() external view returns (uint256);

    function balanceOf(address account) external view returns (uint256);

    function transfer(address recipient, uint256 amount) external returns (bool);

    function mint(address account, uint amount) external;

    function allowance(address owner, address spender) external view returns (uint256);

    function approve(address spender, uint256 amount) external returns (bool);

    function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

    event Transfer(address indexed from, address indexed to, uint256 value);

    event Approval(address indexed owner, address indexed spender, uint256 value);
}

library Address {

    function isContract(address account) internal view returns (bool) {
        bytes32 codehash;
        bytes32 accountHash = 0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470;

        assembly { codehash := extcodehash(account) }
        return (codehash != 0x0 && codehash != accountHash);
    }

    function toPayable(address account) internal pure returns (address payable) {
        return address(uint160(account));
    }

    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");


        (bool success, ) = recipient.call.value(amount)("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }
}

library SafeERC20 {
    using SafeMath for uint256;
    using Address for address;

    function safeTransfer(IERC20 token, address to, uint256 value) internal {
        callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(IERC20 token, address from, address to, uint256 value) internal {
        callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, value));
    }

    function safeApprove(IERC20 token, address spender, uint256 value) internal {
        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    function safeDecreaseAllowance(IERC20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).sub(value, "SafeERC20: decreased
allowance below zero");
        callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowance));
    }

    function callOptionalReturn(IERC20 token, bytes memory data) private {
        require(address(token).isContract(), "SafeERC20: call to non-contract");

        (bool success, bytes memory returndata) = address(token).call(data);
        require(success, "SafeERC20: low-level call failed");

        if (returndata.length > 0) {
            require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
        }
    }
}


contract ExchangePool is Ownable {

    using SafeERC20 for IERC20;
    using SafeMath for uint;

    IERC20 public preCAN;
    IERC20 public newCAN;
    bool private open;

    uint private constant _NOT_ENTERED = 1;
    uint private constant _ENTERED = 2;
    uint public _GENERAL_LOCK; // TEMPORARY: re-entrancy lock guard.

    event Exchange(address account, uint preCanAmount, uint newCanAmount);
    event SetOpen(bool _open);
    event EmergentcyWithdraw(address token, uint amount);

    /// @dev Reentrancy lock guard.
    modifier lock() {
        require(_GENERAL_LOCK == _NOT_ENTERED, 'general lock');
        _GENERAL_LOCK = _ENTERED;
        _;
        _GENERAL_LOCK = _NOT_ENTERED;
    }

    constructor() public{
        preCAN = IERC20(0x1e6395E6B059fc97a4ddA925b6c5ebf19E05c69f);
        newCAN  =  IERC20(0xC95EFaf132507d81805f8Cfb90E4863939310105);                // todo: to be
updated when new can deployed
        open = true;
        _GENERAL_LOCK = _NOT_ENTERED;
    }

    function exchange(uint amount) public checkOpen lock{//knownsec //将旧币兑换成新币
        require(preCAN.balanceOf(msg.sender) >= amount, "preCAN not enough");
        preCAN.safeTransferFrom(msg.sender, address(this), amount);
        newCAN.safeTransfer(msg.sender, amount.mul(200));
        emit Exchange(msg.sender, amount, amount.mul(200));
    }

    modifier checkOpen() {
        require(open, "Pool is closed");
        _;
    }

    function isOpen() external view returns (bool) {
        return open;
    }

    function setOpen(bool _open) external onlyOwner {//knownsec //设置 exchangePool 开放状态
        open = _open;
        emit SetOpen(open);
    }
    function getBalanceOfCAN() public view returns(uint, uint, uint){//knownsec /获取 CAN 代币的旧代币余额、
新代币余额、总代币余额
        uint preCANBalance = preCAN.balanceOf(address(this));
        uint newCANBalance = newCAN.balanceOf(address(this));
        uint totalBalance = preCANBalance.add(newCANBalance);
        return    (preCANBalance, newCANBalance, totalBalance);
    }


    function emergencyWithdraw(IERC20 token, uint amount) public onlyOwner{//knownsec //紧急转账功能
        token.transfer(owner(), amount);
        emit EmergentcyWithdraw(token, amount);
    }
}
```

Unitroller.sol

```
pragma solidity ^0.5.16;

import "./ErrorReporter.sol";
import "./ComptrollerStorage.sol";
/**
 * @title ComptrollerCore
 * @dev Storage for the comptroller is at this address, while execution is delegated to the
`comptrollerImplementation`.
 * CTokens should reference this contract as their comptroller.
 */
contract Unitroller is UnitrollerAdminStorage, ComptrollerErrorReporter {

    /**
      * @notice Emitted when pendingComptrollerImplementation is changed
      */
    event NewPendingImplementation(address oldPendingImplementation, address newPendingImplementation);

    /**
      * @notice Emitted when pendingComptrollerImplementation is accepted, which means comptroller
implementation is updated
      */
    event NewImplementation(address oldImplementation, address newImplementation);

    /**
      * @notice Emitted when pendingAdmin is changed
      */
    event NewPendingAdmin(address oldPendingAdmin, address newPendingAdmin);

    /**
      * @notice Emitted when pendingAdmin is accepted, which means admin is updated
      */
    event NewAdmin(address oldAdmin, address newAdmin);

    constructor() public {
        // Set admin to caller
        admin = msg.sender;
    }

    /*** Admin Functions ***/
    function _setPendingImplementation(address newPendingImplementation) public returns (uint) {

        if (msg.sender != admin) {
            return                                                    fail(Error.UNAUTHORIZED,
FailureInfo.SET_PENDING_IMPLEMENTATION_OWNER_CHECK);
        }

        address oldPendingImplementation = pendingComptrollerImplementation;

        pendingComptrollerImplementation = newPendingImplementation;

        emit NewPendingImplementation(oldPendingImplementation, pendingComptrollerImplementation);

        return uint(Error.NO_ERROR);
    }// knownec  设置候选 Implementation

    /**
    * @notice Accepts new implementation of comptroller. msg.sender must be pendingImplementation
    * @dev Admin function for new implementation to accept it's role as implementation
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
    function _acceptImplementation() public returns (uint) {
        // Check caller is pendingImplementation and pendingImplementation ≠ address(0)
        if (msg.sender != pendingComptrollerImplementation || pendingComptrollerImplementation ==
address(0)) {
            return                                                    fail(Error.UNAUTHORIZED,
FailureInfo.ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK);
        }

        // Save current values for inclusion in log
        address oldImplementation = comptrollerImplementation;
        address oldPendingImplementation = pendingComptrollerImplementation;
        comptrollerImplementation = pendingComptrollerImplementation;

        pendingComptrollerImplementation = address(0);

        emit NewImplementation(oldImplementation, comptrollerImplementation);
        emit NewPendingImplementation(oldPendingImplementation, pendingComptrollerImplementation);

        return uint(Error.NO_ERROR);
    }// knownsec 确定新 Implementation


    /**
      * @notice Begins transfer of admin rights. The newPendingAdmin must call `_acceptAdmin` to finalize the
transfer.
      * @dev Admin function to begin change of admin. The newPendingAdmin must call `_acceptAdmin` to
finalize the transfer.
      * @param newPendingAdmin New pending admin.
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _setPendingAdmin(address newPendingAdmin) public returns (uint) {
        // Check caller = admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_ADMIN_OWNER_CHECK);
        }

        // Save current value, if any, for inclusion in log
        address oldPendingAdmin = pendingAdmin;

        // Store pendingAdmin with value newPendingAdmin
        pendingAdmin = newPendingAdmin;

        // Emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin)
        emit NewPendingAdmin(oldPendingAdmin, newPendingAdmin);

        return uint(Error.NO_ERROR);
    }// knownsec 设置候选 Admin

    /**
      * @notice Accepts transfer of admin rights. msg.sender must be pendingAdmin
      * @dev Admin function for pending admin to accept role and update admin
      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
      */
    function _acceptAdmin() public returns (uint) {
        // Check caller is pendingAdmin and pendingAdmin ≠ address(0)
        if (msg.sender != pendingAdmin || msg.sender == address(0)) {
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_ADMIN_PENDING_ADMIN_CHECK);
        }

        // Save current values for inclusion in log
        address oldAdmin = admin;
        address oldPendingAdmin = pendingAdmin;

        // Store admin with value pendingAdmin
        admin = pendingAdmin;

        // Clear the pending value
        pendingAdmin = address(0);

        emit NewAdmin(oldAdmin, admin);
        emit NewPendingAdmin(oldPendingAdmin, pendingAdmin);

        return uint(Error.NO_ERROR);
    }// knownsec 确定新 Admin

    /**
     * @dev Delegates execution to an implementation contract.
     * It returns to the external caller whatever the implementation returns
     * or forwards reverts.
     */
    function () payable external {
        // delegate all other functions to current implementation
        (bool success, ) = comptrollerImplementation.delegatecall(msg.data);

        assembly {
              let free_mem_ptr := mload(0x40)
              returndatacopy(free_mem_ptr, 0, returndatasize)

              switch success
              case 0 { revert(free_mem_ptr, returndatasize) }
              default { return(free_mem_ptr, returndatasize) }
        }
    }
}
```

6. 附录 B：安全风险评级标准

智能合约漏洞评级标准

漏洞评级	漏洞评级说明
高危漏洞	能直接造成代币合约或用户资金损失的漏洞，如：能造成代币价值归零的数值溢出漏洞、能造成交易所损失代币的假充值漏洞、能造成合约账户损失 HT 或代币的重入漏洞等； 能造成代币合约归属权丢失的漏洞，如：关键函数的访问控制缺陷、call 注入导致关键函数访问控制绕过等； 能造成代币合约无法正常工作的漏洞，如：因向恶意地址发送 HT 导致的拒绝服务漏洞、因 gas 耗尽导致的拒绝服务漏洞。
中危漏洞	需要特定地址才能触发的高风险漏洞，如代币合约拥有者才能触发的数值溢出漏洞等；非关键函数的访问控制缺陷、不能造成直接资金损失的逻辑设计缺陷等。
低危漏洞	难以被触发的漏洞、触发之后危害有限的漏洞，如需要大量 HT 或代币才能触发的数值溢出漏洞、触发数值溢出后攻击者无法直接获利的漏洞、通过指定高 gas 触发的事务顺序依赖风险等。

7. 附录 C：智能合约安全审计工具简介

7.1. Manticore

Manticore 是一个分析二进制文件和智能合约的符号执行工具，Manticore 包含一个符号以太坊虚拟机（EVM），一个 EVM 反汇编器/汇编器以及一个用于自动编译和分析 Solidity 的方便界面。它还集成了 Ethersplay，用于 EVM 字节码的 Bit of Traits of Bits 可视化反汇编程序，用于可视化分析。 与二进制文件一样，Manticore 提供了一个简单的命令行界面和一个用于分析 EVM 字节码的 Python API。

7.2. Oyente

Oyente 是一个智能合约分析工具，Oyente 可以用来检测智能合约中常见的 bug，比如 reentrancy、事务排序依赖等等。更方便的是，Oyente 的设计是模块化的，所以这让高级用户可以实现并插入他们自己的检测逻辑，以检查他们的合约中自定义的属性。

7.3. securify.sh

Securify 可以验证以太坊智能合约常见的安全问题，例如交易乱序和缺少输入验证，它在全自动化的同时分析程序所有可能的执行路径，此外，Securify 还具有用于指定漏洞的特定语言，这使 Securify 能够随时关注当前的安全性和其他可靠性问题。

7.4. Echidna

Echidna 是一个为了对 EVM 代码进行模糊测试而设计的 Haskell 库。

7.5. MAIAN

MAIAN 是一个用于查找以太坊智能合约漏洞的自动化工具，Maian 处理合约的字节码，并尝试建立一系列交易以找出并确认错误。

7.6. ethersplay

ethersplay 是一个 EVM 反汇编器，其中包含了相关分析工具。

7.7. ida-evm

ida-evm 是一个针对以太坊虚拟机（EVM）的 IDA 处理器模块。

7.8. Remix-ide

Remix 是一款基于浏览器的编译器和 IDE，可让用户使用 Solidity 语言构建以太坊合约并调试交易。

7.9. 知道创宇区块链安全审计人员专用工具包

知道创宇渗透测试人员专用工具包，由知道创宇渗透测试工程师研发，收集和使用，包含专用于测试人员的批量自动测试工具，自主研发的工具、脚本或利用工具等。

知道创宇

北京知道创宇信息技术股份有限公司

咨询电话	+86(10)400 060 9587
邮　　箱	sec@knownsec.com
官　　网	www.knownsec.com
地　　址	北京市 朝阳区 望京 SOHO T2-B座-2509