大作业--供应链金融

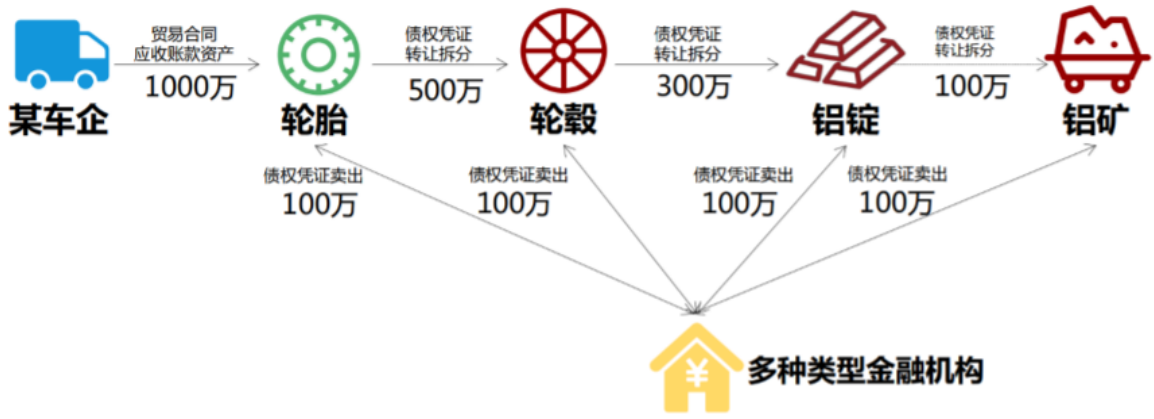
年级	2017	专业（方向）	软件工程
学号	17343008	姓名	陈灿辉
电话	13580900798	Email	1178160567@qq.com
开始日期	2019-11-01	完成日期	2019-12-13

作业内容

项目设计说明：根据提供的供应链场景，基于FISCO-BCOS设计相关的智能合约并详细解释智能合约是如何解决提出的问题。

功能测试文档：将智能合约部署至链上（单节点or多节点），并调用相关函数，详细说明上述的四个功能具体是如何实现的。（截图说明调用结果）

项目背景



传统供应链金融

某车企（宝马）因为其造车技术特别牛，消费者口碑好，所以其在同行业中占据绝对优势地位。因此，在金融机构（银行）对该车企的信用评级将很高，认为他有很大的风险承担的能力。在某次交易中，该车企从轮胎公司购买了一批轮胎，但由于资金暂时短缺向轮胎公司签订了1000万的应收账款单据，承诺1年后归还轮胎公司1000万。这个过程可以拉上金融机构例如银行来对这笔交易作见证，确认这笔交易的真实性。在接下里的几个月里，轮胎公司因为资金短缺需要融资，这个时候它可以凭借跟某车企签订的应收账款单据向金融结构借款，金融机构认可该车企（核心企业）的还款能力，因此愿意借款给轮胎公司。但是，这样的信任关系并不会往下游传递。在某个交易中，轮胎公司从轮毂公司购买了一批轮毂，但由于租金暂时短缺向轮胎公司签订了500万的应收账款单据，承诺1年后归还轮胎公司500万。当轮毂公司想利用这个应收账款单据向金融机构借款融资的时候，金融机构因为不认可轮胎公司的还款能力，需要对轮胎公司进行详细的信用分析以评估其还款能力同时验证应收账款单据的真实性，才能决定是否借款给轮毂公司。这个过程将增加很多经济成本，而这个问题主要是由于该车企的信用无法在整个供应链中传递以及交易信息不透明化所导致的。

区块链+供应链金融

将供应链上的每一笔交易和应收账款单据上链，同时引入第三方可信机构来确认这些信息的交易，例如银行，物流公司等，确保交易和单据的真实性。同时，支持应收账款的转让，融资，清算等，让核心企业的信用可以传递到供应链的下游企业，减小中小企业的融资难度。

实现功能

功能一：实现采购商品一签发应收账款交易上链。例如车企从轮胎公司购买一批轮胎并签订应收账款单据。

功能二：实现应收账款的转让上链，轮胎公司从轮毂公司购买一笔轮毂，便将于车企的应收账款单据部分转让给轮毂公司。轮毂公司可以利用这个新的单据去融资或者要求车企到期时归还钱款。

功能三：利用应收账款向银行融资上链，供应链上所有可以利用应收账款单据向银行申请融资。

功能四：应收账款支付结算上链，应收账款单据到期时核心企业向下游企业支付相应的欠款。

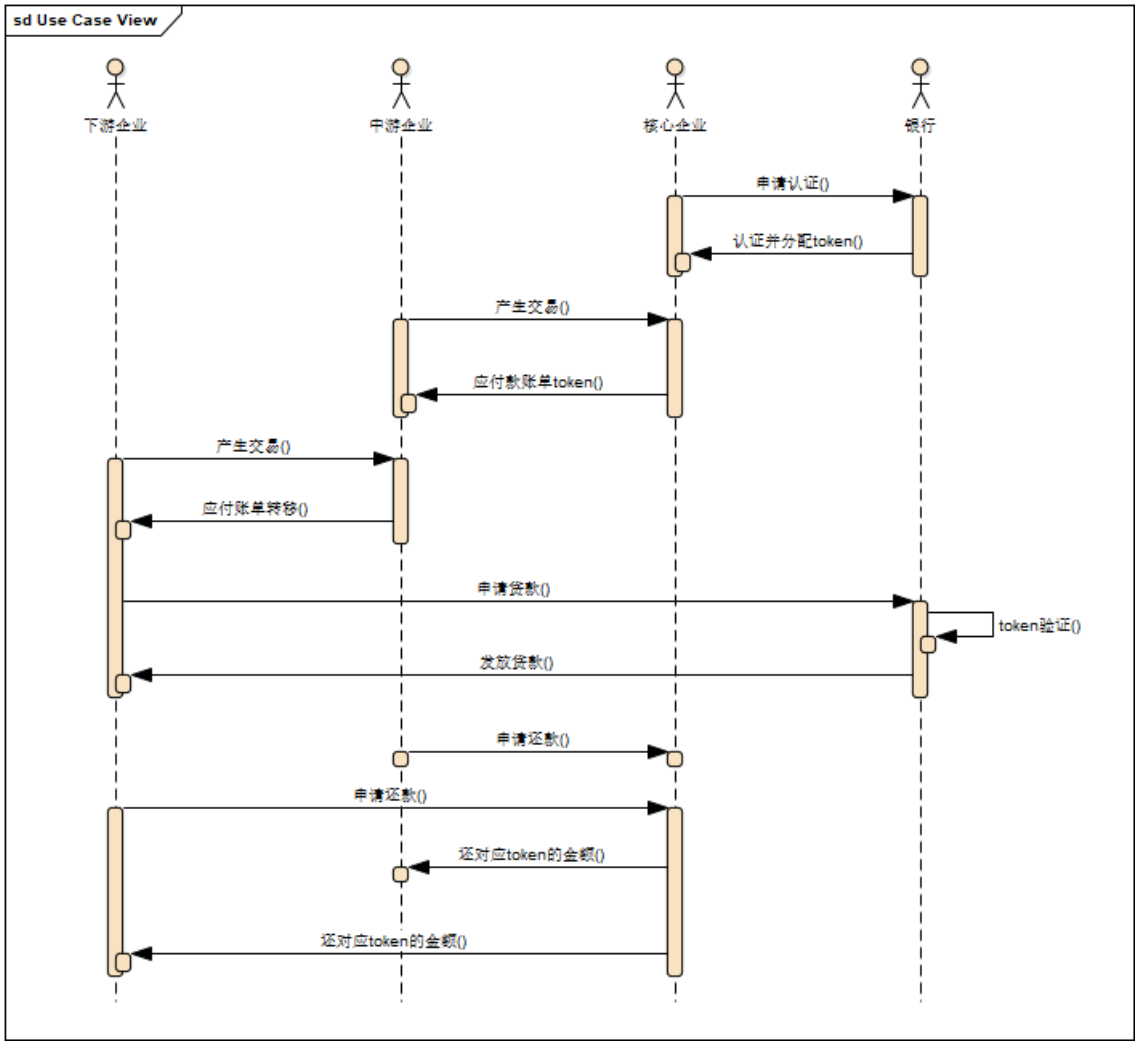
方案设计

设计思想

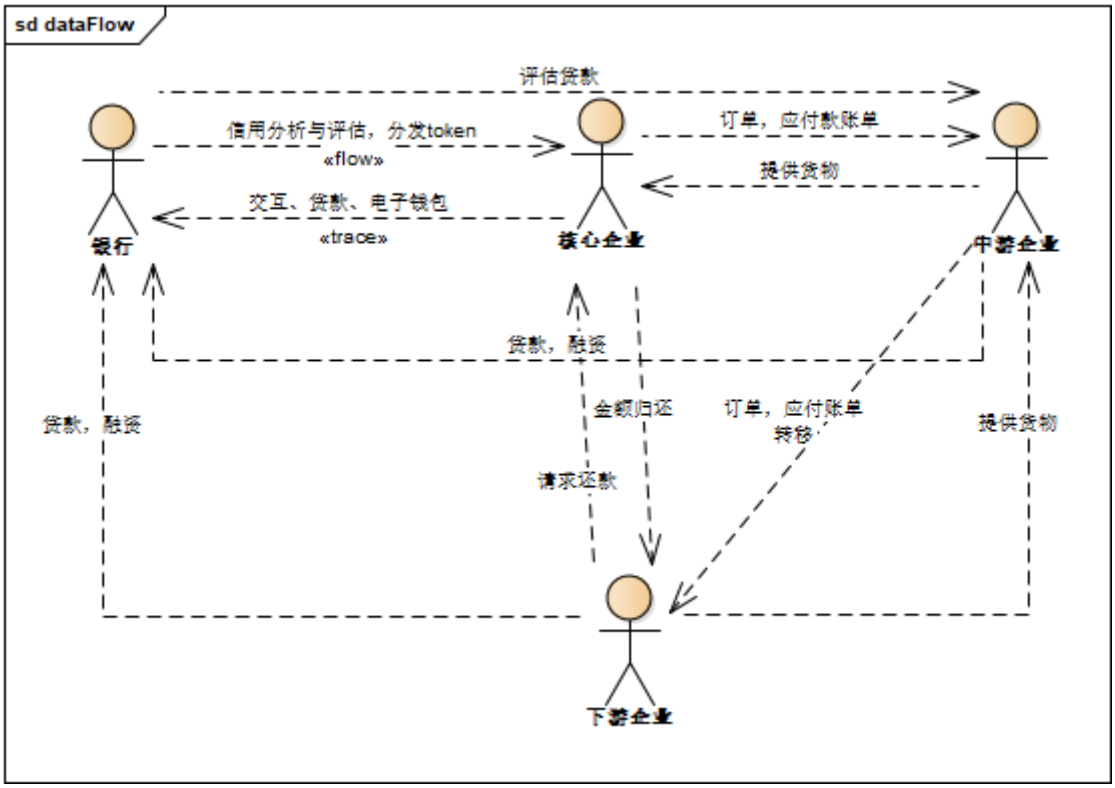
供应链金融的核心是如何处理应付款账单，对此我采用了现在区块链上比较流行的代币的形式，即把企业的应收账单当成是一个企业发行，银行监管的token。企业和银行要对token进行负责，并且认可token的价值。同时为了防止企业无限量或过度发行应收账单，银行需要对核心企业进行评估，即应付账单上应该有发放企业以及银行的联合签名。同时应收账单可以类似token一样流通在网络中，产生交易等。详细的设计细节参见下面的报告。

数据流图

供应链交易流程，这里将供应链上的参与者抽象化成4种角色：银行、核心企业、中游企业、下游企业，他们之间的关系和交互图如下所示



数据流图



存储结构

公司信息

这里我设置的公司信息比较简单，报告公司在链上的账户地址，公司的ID，以及公司名。同时我还做了限制，就是一个账号只能创建一个公司，每一个公司的公司名必须唯一。

```
    struct Company{
        address companyAddr;
        uint companyID;
        string name;
    }
```

发票信息

每一笔的交易都应该记录在发票上面，发票的信息包括其相对应的编号，收/付款方的地址，交易的金额，并且自动写入当前发票成了的时间戳，允许在发票上进行相关的备注

```
    //发票，完成相对应的采购记录后就上链保存
    struct Invoice{
        uint invoiceID; //发票号
        address payeeAddr; //收款方
        address payerAddr; //付款方
        uint amount; //金额
        uint timestamp; //建立的时间戳
        string remark; //相关备注
    }
```

应收账单

核心企业可以通过银行的认证后给下游的企业发行应收账单，应收账单包括付款者的地址，应该归还的时间，以及这笔账单应该归还的本金数

```
    //应收账单
    struct PaymentSchedule {
        address payerAddr; //付款者地址
        uint dueTime; //应还时间，越短越具有即时性,这里我特殊设置0为表示该公司的信用程度，即金融机构最多相信该核心企业的还款能力，对应到就类似于金融机构给核心企业发的token
        uint amount; //应还本金,逐步减少，直到为0的时候就可以删除了
    }
```

贷款信息

贷款信息包括贷款人的账号地址，应该归还的本金，应该归还的利息，以及已经归还的金额

```
    struct LoanInfo{
        address claimerAddr;
        uint principleAmount; //应还本金
        uint interestAmount;  //应还利息
        uint paidAmount; //已还金额
    }
```

电子钱包

这部分是可选的实现，即核心企业可以跟银行合作，使用银行的电子钱包，如网上银行等，与银行的账号自动关联，可以在支付应付款单的时候自动从电子钱包中扣费

```
    //电子钱包-类似网上银行，与银行账号等相关联，直接由银行控制
    mapping (address => uint) moneyBalances;
```

实现功能

权限控制

在这里会涉及到权限的问题，如如果给企业评定登记，贷款业务的处理等，这里都只能由权威的金融机构--银行来进行处理。故我这里先设置了一个抽闲合约，来实现基本的权限管理，这里的权限我简单的设置为owner权限。默认是新建合约的人就是owner，考虑到可能一开始部署合约的是第三方，故在合约中加入了权限转移的函数，使得最终链上的相关认证权利落银行手中，详情请参见`ownable.sol`文件

```
pragma solidity ^0.4.24;

// 基础的权限控制
contract Ownable {
    address private currentOwner;

    event TransferOwnership(address indexed oldOwner, address indexed newOwner);

    // internal只能被继承
    constructor () internal {
        currentOwner = msg.sender;
        emit TransferOwnership(address(0), currentOwner);
    }

    //当前的owner
    function owner() public view returns (address) {
        return currentOwner;
    }

    // a function modifier 'onlyOwner' 添加权限
    modifier onlyOwner() {
        require(msg.sender == currentOwner);
        _;
    }


    // 所有权转移，所有权主要是在银行等金融机构中
    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        emit TransferOwnership(currentOwner, newOwner);
        currentOwner = newOwner;
    }
}
```

公司注册

在SupplyChain.sol文件中，我添加了公司注册等功能，这里公司的相关信息如下所示

```
    struct Company{
        address companyAddr;
        uint companyID;
        string name;
    }

    uint companyNum;
    mapping (uint => Company) companies;
```

每一个账户只允许创建一个公司，并且公司名不能相同

```
    //注册公司
    function registerCompany(string companyName) public returns(uint) {
        address companyAddr = msg.sender;
        for (uint i = 1; i <= companyNum; i++){
            require(!equalTo(companyName, companies[i].name), "Company name
already exists");
            require(companyAddr != companies[i].companyAddr, "Company address
already exists");
        }
        uint companyID = ++companyNum;
        companies[companyID] = Company(companyAddr, companyID, companyName);
        // emit CompanyID(companyID, name);
        return companyID;
    }
```

允许在外部查看当前公司的相关信息，获得连上参加的公司的数量等

```
    function getCompany(uint id) public view returns (address companyAddr, uint
companyID, string name){
        require(id > 0 && id <= companyNum);
        return (companies[id].companyAddr, companies[id].companyID,
companies[id].name);
    }

    function getCompanyNum() public view returns (uint){
        return companyNum;
    }
```

交易上链

首先在产生交易的时候，需要交易双方开具发票，并将相对应的发票上链，保存起来，以便后续进行查询等相关操作

```
    //发票，完成相对应的采购记录后就上链保存
    struct Invoice{
        uint invoiceID; //发票号
        address payeeAddr; //收款方
        address payerAddr; //付款方
        uint amount; //金额
        uint timestamp; //建立的时间戳
        string remark; //相关备注
    }

    uint invoiceNum;
    mapping (uint => Invoice) invoices;
```

这里为了防止一些虚假的发票产生，我这里事先约定，在创建发票的时候应该由付款方主动发起，并将发票的相关信息自动上链保存

```
    //创建发票，应该由付款方主动发起
    function createInvoice(address payeeAddr, address payerAddr, uint amount,
string mark) public onlyOwner(payerAddr) returns(uint){
        uint invoiceID = ++invoiceNum;
        invoices[invoiceID] = Invoice(invoiceID, payeeAddr, payerAddr, amount,
now, mark);
        return invoiceID;
    }
```

银行认证

这部分是银行特有的权限，银行也要为自己的行为负责。首先核心企业可以向银行发起审查的请求，银行可以对企业进行审查，审查其的信用额度，还款能力等，进而允许企业发行一定量的应收账单，这部分应收账单是银行承认的。

企业手中的应收账单存储结构如下

```
    //企业手中拥有的应收账单
    mapping (address => PaymentSchedule []) paymentBalances;
```

而初始的额度分配是由银行完成的，别的机构没有权限

```
    //权威机构通过评估，预先分配token，这里的token类似信用评级
    function distributeToken(address _to, uint _value) public onlyOwner{
        paymentBalances[_to].push(PaymentSchedule(_to, 0, _value));
    }
```

这里我加上银行认证的目的是，让银行也在其中作为应收账单的承担者，额度限定也避免了企业疯狂开应收账单然后跑路的行为。

此外这里我设置dueTime=0表示默认的，及企业内部的可以发放给下游企业的应收账单额度，其作用类似企业发行的，

电子钱包

核心企业可以跟银行之间建立电子钱包，以处理应收账单等开销

```
    //电子钱包-类似网上银行，与银行账号等相关联，直接由银行控制
    mapping (address => uint) moneyBalances;
```

由于电子钱包的服务是由银行提供，存取款的服务必须由银行提供

```
    //存款，取款，必须通过银行
    //存款操作
    function depositMoney(address _to, uint _value) public onlyOwner{
        moneyBalances[_to] += _value;
    }
    //取款操作
    function withdrawMoney(address _addr, uint _value) public onlyOwner{
        require(moneyBalances[_addr] >= _value);
        moneyBalances[_addr] -= _value;
    }
    //yve chaxun
    function getMoneyBalances() public view returns(uint) {
        return moneyBalances[msg.sender];
    }
```

核心企业发放应收账单

核心企业可以将前面银行审批的额度作为其的token，发行应收账单给下级或其交易企业，在发行应收账单的时候要注明时间权限

```
    //核心企业给下级企业分发的应收款单据
    function createPaymentSchedule(address _to, uint _dueTime, uint _amount)
public{
        //找到dueTime = 0的，也就是自己credit能给出的最大应收款单据
        uint len = paymentBalances[msg.sender].length;
        uint i = 0;
        for (; i < len; i++){
            if (paymentBalances[msg.sender][i].dueTime == 0){
                break; //找到了~
            }
        }
        require(i < len, "Your company does not register in the bank"); //没权
限，金融机构信不过
        require(paymentBalances[msg.sender][i].amount > _amount, "Your company
does not have enough credit to make such payment");//有权限，但是不足
        paymentBalances[msg.sender][i].amount -= _amount;
        paymentBalances[_to].push(PaymentSchedule(msg.sender, _dueTime,
_amount));
    }
```

应收账单的转让

企业得到了核心企业发放的应收账单后，在自己企业余额不足的情况下，可以将核心企业发放的应收账单以代币的形式发给相关的交易企业，充当金额的作用。注意到一个企业可能会得到多个应收账单，故可以选择把多个企业的应收账单整合到足够的金额后再转发过去。在前面的数据结构涉及中，我把每一个独立的应收账单看成是一个块来进行处理，故这里就会涉及到块的拆分与合并问题。

```
    //把相对应的应收账单当成是token转账操作
    function transferPayment(address _to, uint _value) public{
```

```
        uint totalAmount = getBalance(msg.sender);
        require(totalAmount > _value); //要有足够的钱呀
        uint currentSum = 0;
        for (uint i = 0; i < paymentBalances[msg.sender].length; i++){
            currentSum += paymentBalances[msg.sender][i].amount;
            PaymentSchedule payment = paymentBalances[msg.sender][i];
            if (currentSum <= _value){
                deletePayment(_to, i);
                paymentBalances[_to].push(payment);
                if (currentSum == _value) break;
            }
            else if (currentSum > _value){ //别转这么多，转块的一部分给你就好了
                uint blockTransferAmount = _value - (totalAmount -
paymentBalances[msg.sender][i].amount);
                paymentBalances[msg.sender][i].amount -= blockTransferAmount;
                //对方多了对应的一个块
                paymentBalances[_to].push(PaymentSchedule(payment.payerAddr,
payment.dueTime, blockTransferAmount));
                break;
            }
        }
    }
```

企业应收账单的查询

由于我把应收账单设计成token的模式，那么企业管理的金额就有以下类型

1. 自己企业内部的金额，这部分不用上链，企业内部自己规划
2. 使用银行的电子钱包，电子钱包是与银行账号等消息相关联的，可以用于付款等服务，这部分是对于中下游企业是可选的
3. 企业手中的应收账单，这部分类似token的功能，核心企业承认，银行也承认

为了方便管理，我设计了查询当前应收账单总额的操作。一总是统计总的账单，同时为了管理的方便，还设置了一个根据时间限制查询总额的操作

```
    //获得它的应收账款
    function getBalance(address addr) public view returns(uint){
        uint len = paymentBalances[addr].length;
        uint sum = 0;
        for (uint i = 0; i < len; i++){
            sum += paymentBalances[addr][i].amount;
        }
        return sum;
    }

    //获得在due time前得收到款的payment
    function getBalance(address addr, uint dueTime) public view returns(uint){
        uint len = paymentBalances[addr].length;
        uint sum = 0;
        for (uint i = 0; i < len; i++){
            if (paymentBalances[addr][i].dueTime <= dueTime){
                sum += paymentBalances[addr][i].amount;
            }
        }
        return sum;
    }
```

可以通过调用getPaymentSchedule函数来获得每一笔应收账单的详细信息

```
    function getPaymentSchedule(address _addr, uint _paymentID) public view
returns(address payerAddr_, uint dueTime_, uint amount_){
        PaymentSchedule storage ps = paymentBalances[_addr][_paymentID];
        require(_paymentID >= 0 && _paymentID < paymentBalances[_addr].length,
"out of range");
        return (ps.payerAddr, ps.dueTime, ps.amount);
    }
```

融资贷款服务

企业可以想银行申请贷款服务，银行可以通过审查企业的信用及其实际的还款能力，参考其现拥有的应收货款，决定给他贷款的额度

这里我将贷款的申请与响应封装成相对应的事件

```
    event OnLoadRequestEvent(address requester, uint requestAmount);
    event OnLoadResponseEvent(address requester, uint requestAmount, bool
result);
```

企业可以对银行发起贷款融资的请求

```
    // 用户调用这个来对核心企业发出付款请求
    function requestForPayment(uint paymentID) public{
        require(paymentID >= 0 && paymentID <
paymentBalances[msg.sender].length);
        PaymentSchedule payment = paymentBalances[msg.sender][paymentID];
        require(now >= payment.dueTime);
        emit OnPaymentRequestEvent(msg.sender, payment.payerAddr, paymentID);//
触发事件
    }
```

银行在审查企业的相关信息后可以决定是否接受其贷款

```
    //接收并发布贷款信息
    function acceptLoan(address requesterAddr, uint requestAmount) public
onlyOwner{
        loans.push(LoanInfo(requesterAddr, requestAmount, 0, 0));
        emit OnLoadResponseEvent(requesterAddr, requestAmount, true);
    }

    //拒绝贷款信息
    function rejectLoan(address requesterAddr, uint requestAmount) public
onlyOwner{
        emit OnLoadResponseEvent(requesterAddr, requestAmount, false);
    }
```

此外，银行可以选择自动处理，比如申请的企业手中的应收账单的总额大于其申请的额度的时候，就可以默认允许他申请贷款融资

```
    function responseLoan(address requesterAddr, uint requestAmount) public
onlyOwner{
        //获得它应收款数量
        uint sum = getBalance(requesterAddr);
        if (sum >= requestAmount){
            //收款数足够多，那么我们相信有还款能力
            loans.push(LoanInfo(requesterAddr, requestAmount, 0, 0));
            emit OnLoadResponseEvent(requesterAddr, requestAmount, true);
        }
    }
```

应收账单还款

我将企业应收账账单的还款等设计一个事件，在申请还款服务，以及还款的时候会触发相对应的事件

```
    event OnPaymentRequestEvent(address requester, address payer, uint
paymentID);
    event OnPaymentResponseEvent(address requester, address responer, uint
paymentID);
```

相关企业可以向核心企业发出付款的请求

```
    // 用户调用这个来对核心企业发出付款请求
    function requestForPayment(uint paymentID) public{
        require(paymentID >= 0 && paymentID <
paymentBalances[msg.sender].length);
        PaymentSchedule payment = paymentBalances[msg.sender][paymentID];
        require(now >= payment.dueTime);
        emit OnPaymentRequestEvent(msg.sender, payment.payerAddr, paymentID);//
触发事件
    }
```

企业在接收到相对应的请求后，审查应收账单上面是否是自己的签名，如果是的话，就完成相对应的还款服务，同时企业的credit就相对应的增加，他能发放的应收账单的额度也增加回来，这就保证了企业可以多次发放应收账单

```
    //针对某一个具体的payment发起付款
    function payForPaymentSchedule(address _to, uint paymentID, uint _amount)
public{
        require(paymentID >= 0 && paymentID < paymentBalances[_to].length);
        require(msg.sender == paymentBalances[_to][paymentID].payerAddr); //确实
应该是我付款
        require(_amount <= paymentBalances[_to][paymentID].amount); //当然不会多付
款
        require(_amount <= moneyBalances[msg.sender]);
        if (_amount == paymentBalances[_to][paymentID].amount){
            deletePayment(_to, paymentID);
        } else {
            paymentBalances[_to][paymentID].amount -= _amount;
        }
        //我直接的信用就回来了
        uint id = getSelfCreditPayment(msg.sender);
        if (id < paymentBalances[msg.sender].length){
            paymentBalances[msg.sender][id].amount += _amount;
        }
```

```
        moneyBalances[msg.sender] -= _amount;//账号上的前少了
        moneyBalances[_to] += _amount;
        emit OnPaymentResponseEvent(_to, msg.sender, paymentID);
    }
```

功能测试

创建用户

这里为了测试，创建如下私钥用户

用户名称	用户ID	用户描述	用户公钥地址信息	用户状态	操作
轮毂公司	700007	供应链下游企业	0x472612296c99ac410bfec...	正常	修改
轮胎公司	700006	供应链中游企业	0x824dd0aad6c16351080c9...	正常	修改
宝马	700005	核心企业	0x5329a34b5ad59989d0c5...	正常	修改
Bank	700004	金融机构--银行	0x2b103db4455630ca18ea...	正常	修改

其中：

Bank为银行，为供应链中的金融机构

宝马：为供应链中的核心企业，银行对该企业的信用等级评级很高，认为其有有很大的风险承担的能力

轮胎公司：为供应链中的中游企业，与宝马公司打交道

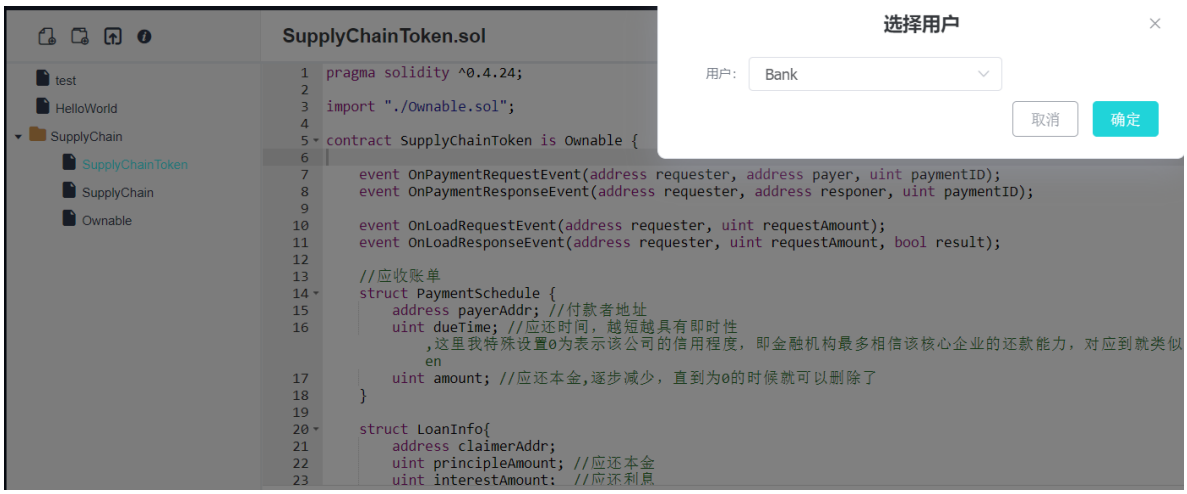
轮毂公司：为供应链中的下游企业，与轮胎公司打交道

本次测试创建的用户名，及其公钥地址信息如下所示

用户名	账号地址
Bank	0xbd86bd51ba8c2c656c7fa375d6776e11e815c544
宝马	0xfe2333260df1e39f5884c19e7b3a4c246e09e780
轮胎公司	0x354e74d5b83ca5f8bdddd489052f6edf5ea5677f
轮毂公司	0x7ab122c8eeee530ac4bbdbe8e38d2f608d1d7837

合约部署

首先应该有银行金融机构部署SupplyChainToken， SupplyChain合约，以保证其对合约的所有权



如果是别的机构先部署合约，那么需要合约的构建者需要调用transferOwnership函数，将合约的所有权转移给银行或相关的金融机构，以保障链上的公平性

```
    /// Define a public function to transfer ownership
    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        emit TransferOwnership(currentOwner, newOwner);
        currentOwner = newOwner;
    }
```

部署合约后，合约列表中应该有如下2个合约

合约名称	合约目录	合约地址	合约abi	合约bin	创建时间	操作
SupplyChain	SupplyChain	0x1c24e950e5554f7e834678c4358e...	[{"constant":false,"inputs":[{"name":"c...	6080604052600436106100615760...	2019-11-13 08:19:31	发送交易
SupplyChainToken	SupplyChain	0x5d67eef8aa1bbe52c3d753f4bd80...	[{"constant":true,"inputs":[{"name":"_...	6080604052600436106101075760...	2019-11-13 08:19:56	发送交易

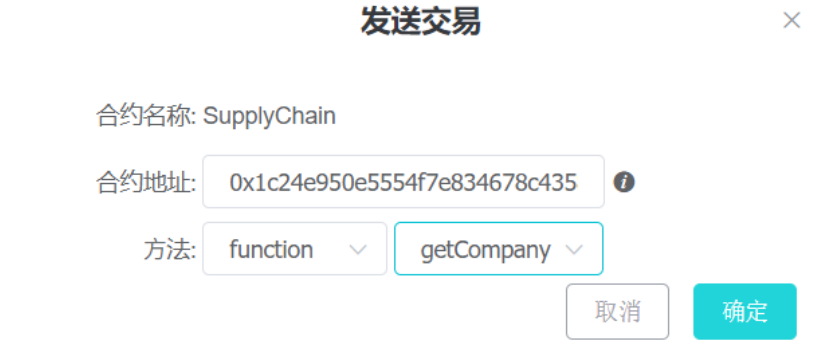
公司注册

公司的相关信息可以注册在区块链上面，以便链上的企业进行查询等操作，如下为银行的注册

以下先为各个公司进行注册操作，如下为轮胎公司，其余公司注册类似，这里就不在赘述

这里注意到，我们要求一个账号只能对应于一个公司，同时为了区分公司名必须是独立且唯一的，在注册的时候就会进行相对应的认证服务，如下，如果一个账号重复注册公司，或者注册同名公司的话，就会导致交易失败

此外还可以通过SupplyChain合约查询链上注册的公司数量



还可以通过公司注册时返回的公司号查询公司的相关信息，如下为查询1号公司，也就是银行的信息

可以看到得到以下的结果

copy


```
▶{
  ▶resData: [
      "0x2b103db4455630ca18ea729a48b36234c0f4fca9",
      1,
      "银行"
  ],
```


银行认证

首先企业可以到银行中完成相对应的认证任务，由银行对企业的信用进行分析，并且进一步评估其还款能力，评估后，根据其资产及其还款能力。评估后由银行对相对应的企业分发相对应数量的token，表示银行认可的，其最大可以发行的应收账款

这里由于宝马公司为核心企业，银行认为其有较大的还款能力，这里给宝马企业分发了1000个token，表示银行将承认其有1000（万）的还款能力，故也能分发1000（万）的应收账款

银行分发Token给宝马公司的交易如下

发起交易后，查询宝马公司的token数量

根据返回的值可以看到，宝马公司目前确实有1000个token


```
{
  resData: [
    1000
  ],
```


值得一提的是，银行认证服务只能由银行发起，其他账号是无权发起的，这样就保障了安全性。如下，轮胎公司尝试给自己发1000个token

那么就会导致交易失败，啥也得不到，同时这种非法操作也会被记录在区块链上，留下不良的记录呀

电子钱包服务

除了上面应收账款对应的token外，合约中还有相对应的电子钱包的服务，其是由银行直接提供的类似“网上银行”的服务，其电子钱包中的钱对应的就直接是银行中的钱啦。当然，存款和取款操作，也只能由银行进行相对应的操作

如下为宝马公司在银行的电子钱包中存款1000的操作

当然有存款也可以取款，也就是将金额从电子钱包中提取出来，存取款为核心业务，只能由银行进行操作，同时存取款时会检查金额与余额，如果不满足条件就会报错

如下如果尝试从宝马公司的账号中提取出1001的存款的话就会报错，因为~他只存了1000元呀

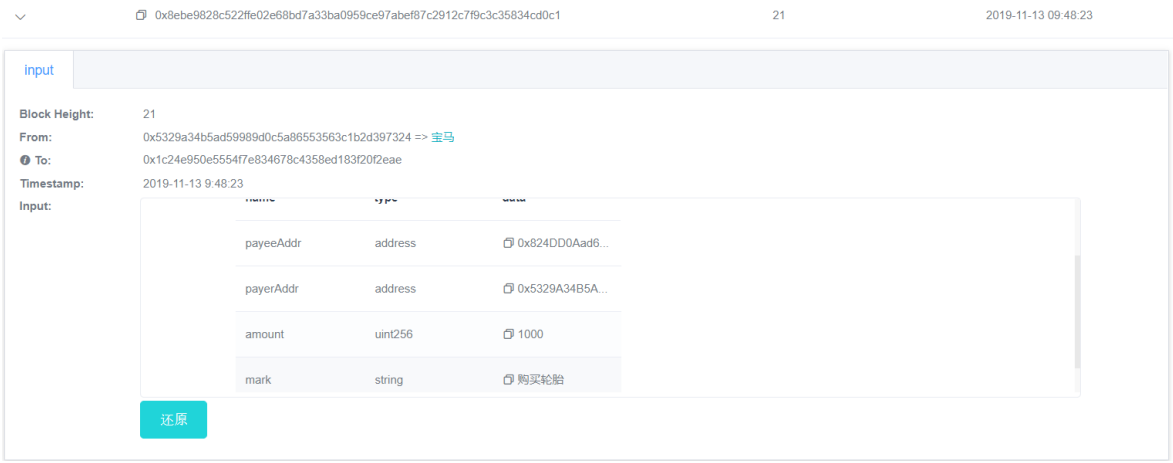
如下，会出现交易报错的信息

功能一、交易上链

功能一：实现采购商品—签发应收账款 交易上链。例如车企从轮胎公司购买一批轮胎并签订应收账款单据。

首先通过SupplyChain合约，在链上可以发布发票，如下宝马公司向轮胎公司购买轮胎，我们可以在链上发布发票，操作如下所示

发布之后就可以在链上查看到相关的消息记录



假设此时于宝马公司由于资金暂时短缺向轮胎公司签订了 1000 万的应收账款单据，承诺 1 年后归还轮胎公司 1000 万，那么我们可以在SupplyChainToken合约中发布相关的应收账款给轮胎公司

如下，这里为了测试发布将dueTime设置为10s

发布交易成功后如下所示

此时查询轮胎公司的应收账单的金额，可以看到轮胎公司的应收账单的金额已经增加了

变成了1000的应收账单

交易内容

copy


```
{
  resData: [
    1000
  ],
  input: {
    name: "getBalance",
    inputs: [
```

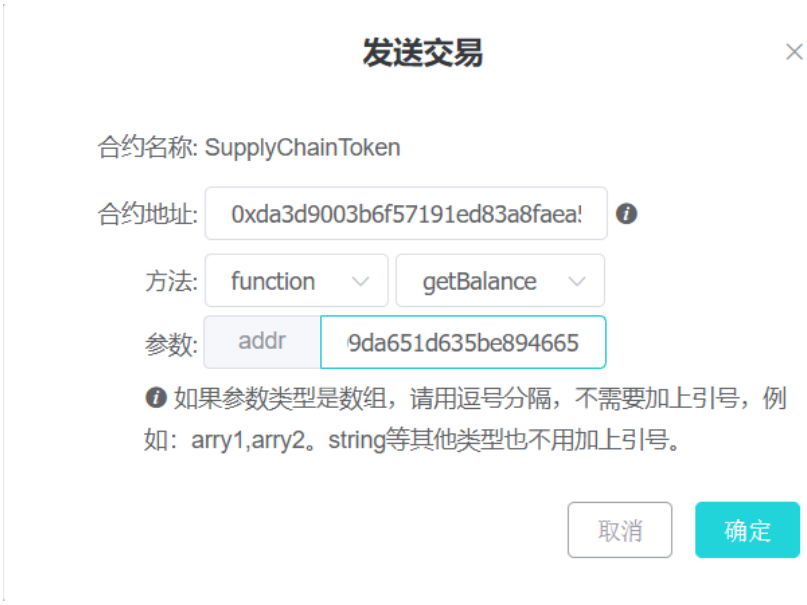

功能二、应收账单的转让

功能二：实现应收账款的转让上链，轮胎公司从轮毂公司购买一笔轮毂，便将于车企的应收账款单据部分转让给轮毂公司。轮毂公司可以利用这个新的单据去融资或者要求车企到期时归还钱款。

首先轮胎公司从轮毂公司购买一笔轮毂，该交易记录应该生成发票并且上链，发票生成的操作如下所示

发票上链后，轮胎公司可以将车企的应收账单部分转让给轮毂公司，对应的操作如下所示

转账成功后，轮胎公司应收账款余额就减少了，余额查询如下所示



可以看到，它的应收账款变成了500

copy

```
{
  resData: [
    500
  ],
```

同时轮毂公司的应收账款增加，查询结果如下所示

copy

```
{
  resData: [
    500
  ],
```

查看详细信息后可以看到，轮毂公司的应收账款的来源正是轮胎公司

copy


```
{
  resData: [
    "0x5329a34b5ad59989d0c5a86553563c1b2d397324",
    10,
    500
  ],
```


功能三、利用应收账款向银行融资

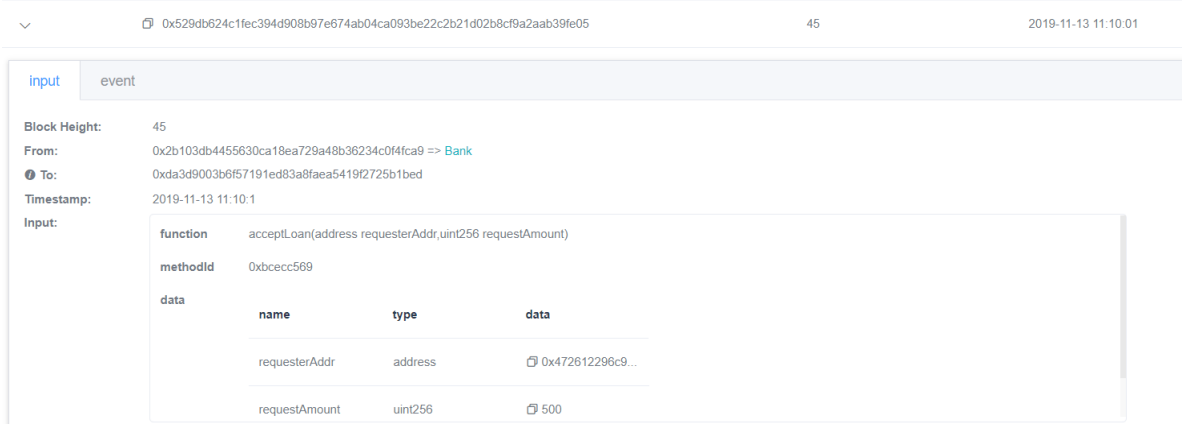
如果轮毂公司向银行申请1000的贷款进行融资

由于银行查询到其应收账款只有500，故银行可以拒绝其的贷款请求

如轮毂公司向银行申请500的贷款进行融资

由于银行查询到该公司确实有500的应收账款，故认为该公司有能力偿还贷款，所以就接受他的融资请求

相对应的贷款请求也会被保存在区块链上



功能四、支付应收账款

到期后轮胎公式可以要求车企还款

宝马公司收到相对应的通知后，可以对当初的应收账款进行还款操作

还款后，相对应的应收账款应该被收回撤销，此时查询轮胎厂的应收账单发现，其的应收账单已经被车企回收回去，以避免double spending的问题


```
▶ {
  ▶ resData: [
      0
  ],
```


界面展示

登录界面

登录界面如下所示，虽然区块链上面的账号主要是通过address和私钥来作为账号的，但是这样并不用户友好，因为在本地登录的时候如果只用记住自己的账号密码就好了。而账号对应的address等则是由后端完成的。

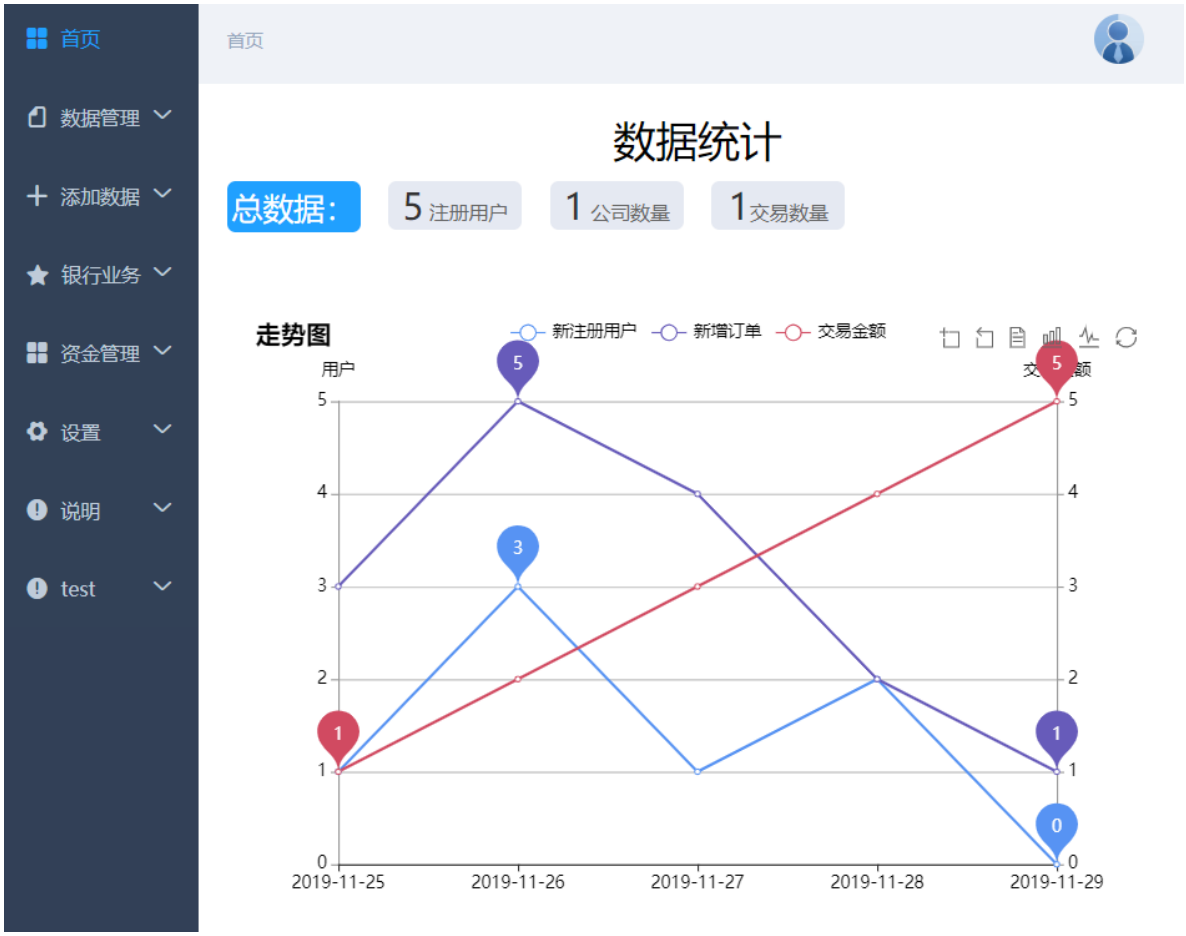
注册界面

我是使用Vue实现的单页面的操作，注册界面如下，注册时主要是登记用户名，密码等信息，然后就会在Fisco Bcos后端创建一个新的账号，生成新的公私钥密码等存储在后端。注册后，用户可以根据用户名和密码直接登录进入系统。由于是本地存储，账号密码等信息存储在本地后端的user.json文件夹中

首页

登录之后会进入到首页，如下所示。

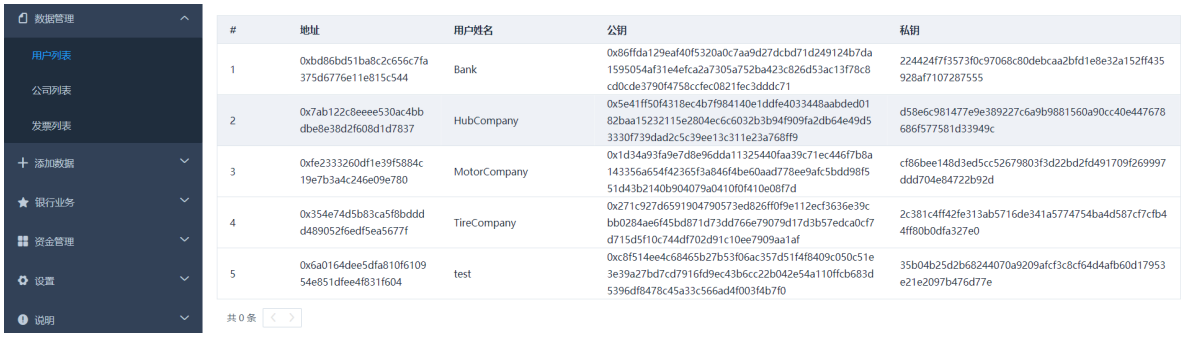
左边是侧旁框，其包含了主要的功能。而首页主要展示了供应链上的相关信息，如公司数量，注册用户，交易数量等信息。



数据管理

用户列表

用户列表的功能主要展示了当前联盟链上的本地管理的地址信息，公钥，私钥等信息，这里信息的可以方便后面的数字签名认证，转账等操作。



公司列表

公司列表展示了联盟链上注册的公司信息

点击小箭头还可以进一步查看公司的详细信息

如果当前用户是该公司的注册者，也就是账号地址所对应的用户的话，那么就可以注销公司

发票列表

发票列表主要展示了当前链上的发票交易等信息，链上的公司可以将发票上链后这里查看，同样的点击每一行中的小箭头可以展开更多信息进行查看。

添加数据

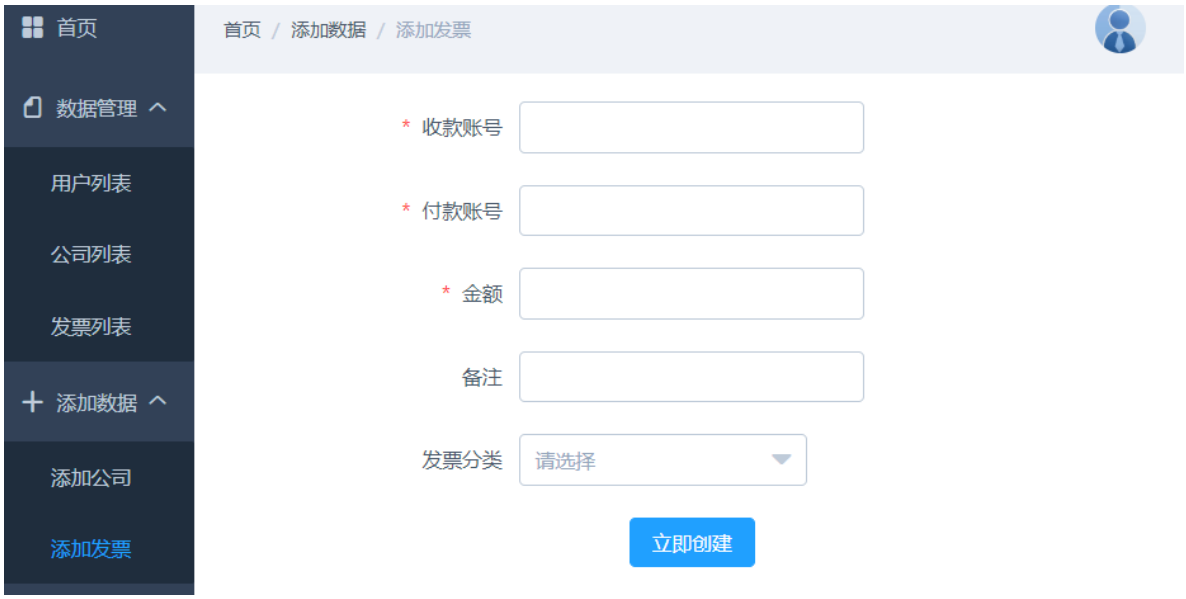
添加公司

添加数据部分主要是可以注册添加信息，并且广播到链上，如下为添加公司，也就是注册公司的信息，点击立即创建后就可以注册公司了。注意到注册失败的原因可能如下：

1. 一个账号只能创建一个公司，不允许仿佛多次创建公司
2. 创建公司的时候要注意，公司名称是不能重复的

添加发票

添加发票的页面如下所示，发票主要是用来见证双方的交易记录的。注意到，为了防止有企业创建虚假发票，这里做了一个限制，就是当前用户的账号必须是付款账号，否则就会创建失败。这样就防止了有公司将其他公司作为付款账号，而将自己作为收款账号，从而破坏整一个链上的金融安全性。



银行业务

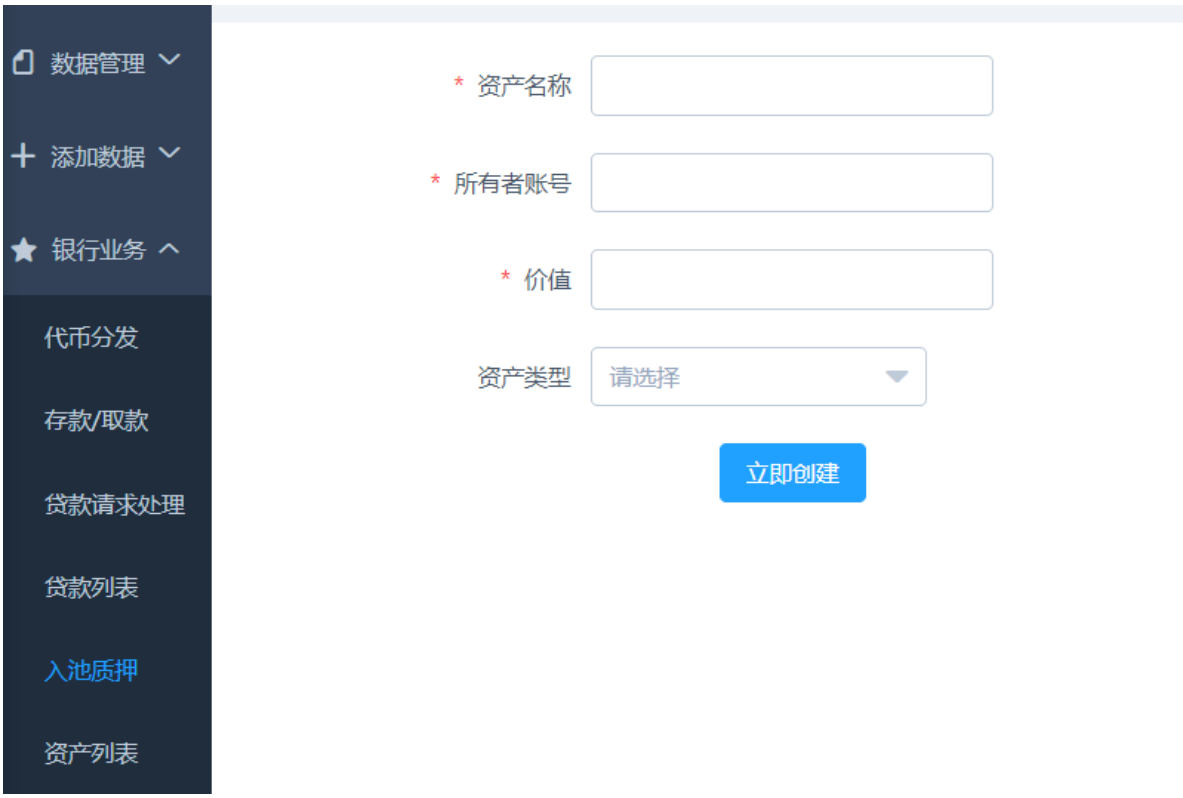
银行业务的权限仅仅限制为银行或权威的金融机构，这部分功能非银行的账户是没有权限操控使用的。也就是进行操作的时候会报错。

入池质押

这部分主要对应到的是供应链上入池质押融资部分

页面展示如下

需要填写资产名称和所有者账号，这两部分就唯一确定了资产池中的唯一资产。进行资产注册 或质押后，企业可以得到对应的融资或者银行分发的token，从而有分放应收账款的额度。



代币分发

代币分发，这里的代币token其本质代表的就是银行对企业的付款能力的认证。代币分发也供应链上金融机构（银行）的核心业务之一。银行需要对企业的还款能力和历史信誉进行考量，根据相关信息对核心企业分发token，分发的token会带上银行的签名。

存款/取款

这部分也是银行的主要功能，这里主要是涉及到电子钱包的概念。类似网上银行的概念，企业可以选择将自己账户上的一些资金以电子钱包的概念存储在银行中，该电子钱包直接对应到现实生活中的金钱。该核心业务也是由银行把控。电子钱包的前可以用来直接支付应收账款，进行资产鉴定等。

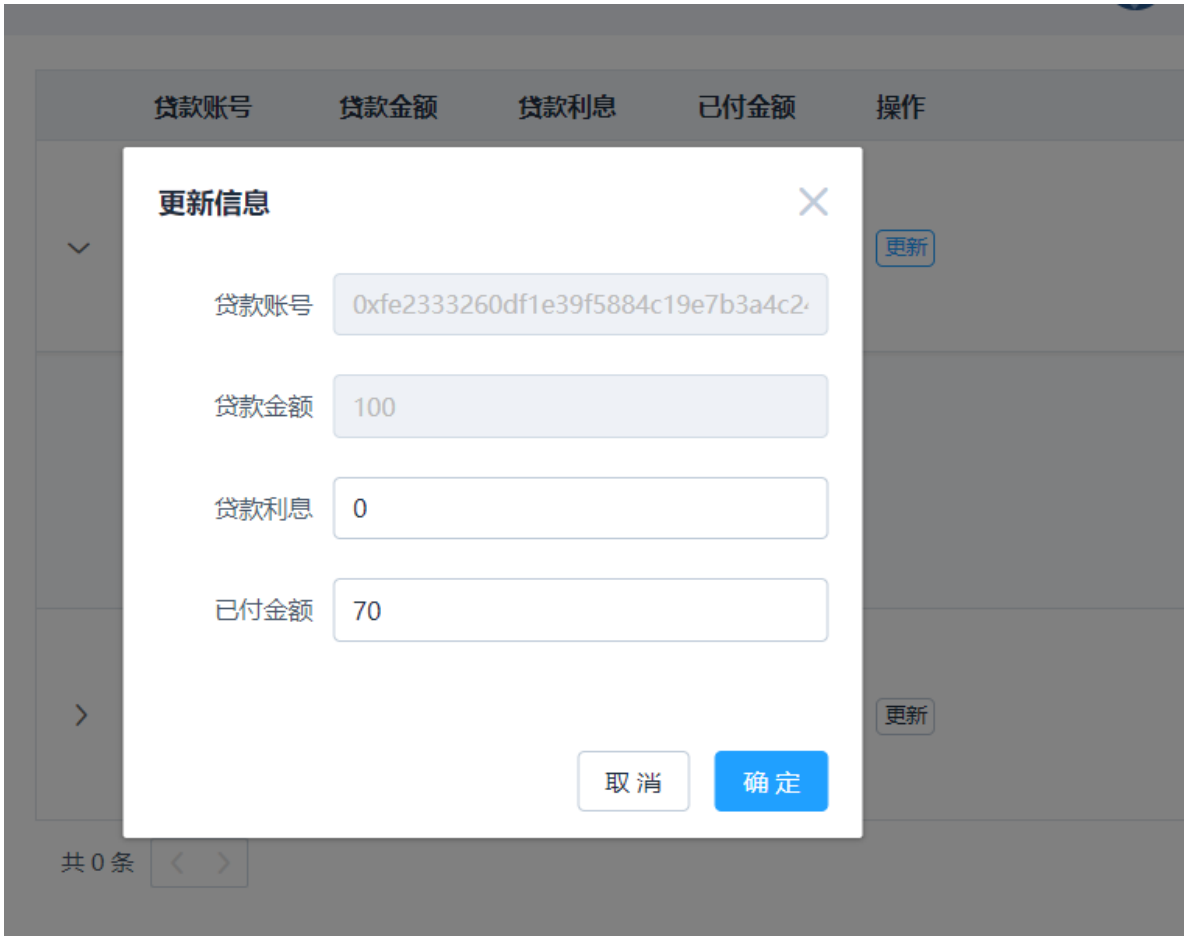
贷款请求处理

企业向银行发送贷款请求后，对应的贷款请求就会出现在该表中。银行可以根据企业的信用额度，流通资金数，应收账款等信息进行判断，判断企业是否偿还贷款的能力，进一步确定是否接受贷款，或者是直接打回。

贷款列表

银行如果同意企业贷款后，贷款的相关信息会出现在贷款列表中，包括贷款的账号，贷款的金额，利息，以及已经偿还的金额数。

随着企业借贷款的时间延长，贷款的利息也会随之增加。同时企业可能会偿还贷款的时候，企业的已付金额也要得到更新。需要更新数据的时候点击更新按钮就可以了。贷款信息更新界面如下所示



资产列表

在入池质押页面上登记资产后，如下登记工厂的资产

然后就可以在资产列表中查找到相对应的资产项，如下所示

如果企业要推出联盟链，或者资产审批不合格，企业申请注销资产的话，可以点击注销按钮，从而注销已经注册的资产

资金管理

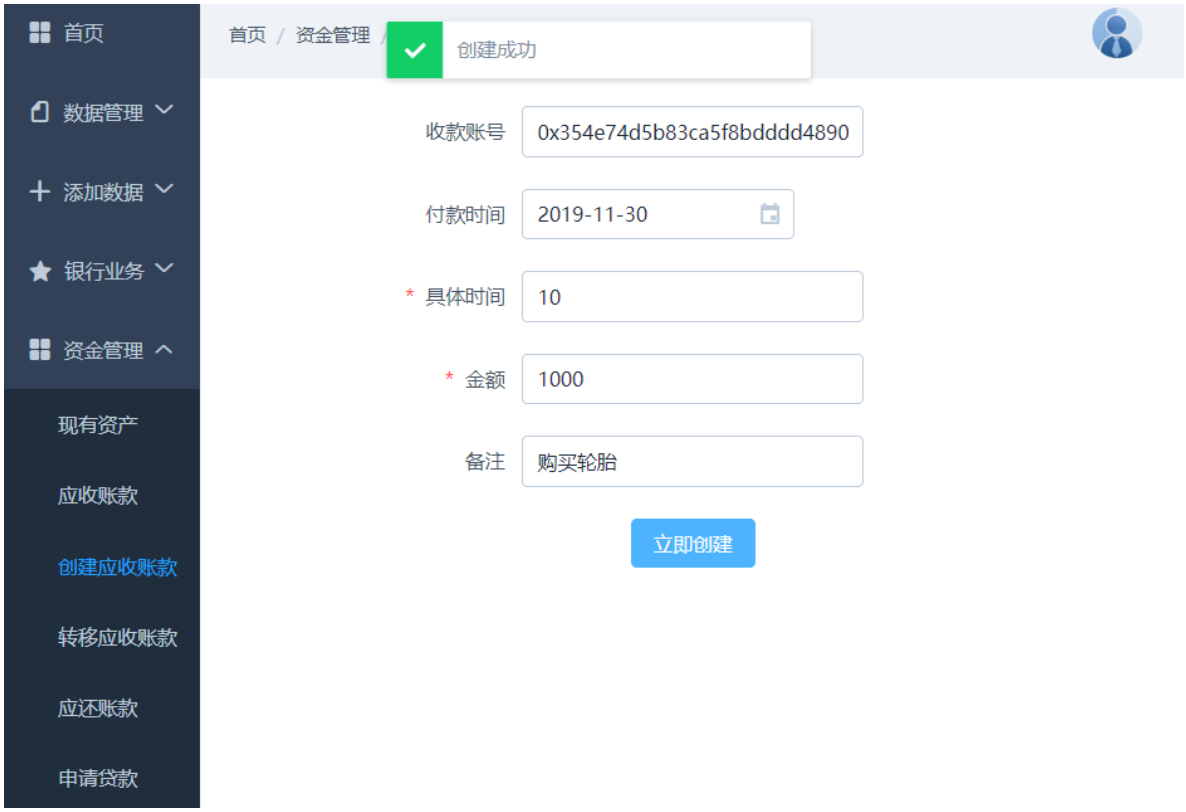
资金管理包括的功能是联盟链上各个企业的主要操作，一些操作与企业的资产相关

现有资产

现有资产主要显示当前账号所属资产，主要包括电子钱包（银行提供的网上银行的功能），主要来源是转账操作或者是银行存款操作；还有就是应收账款，其中对应核心企业来说其应收账款包括一部分初始银行认证的，分发给核心企业的应收账款

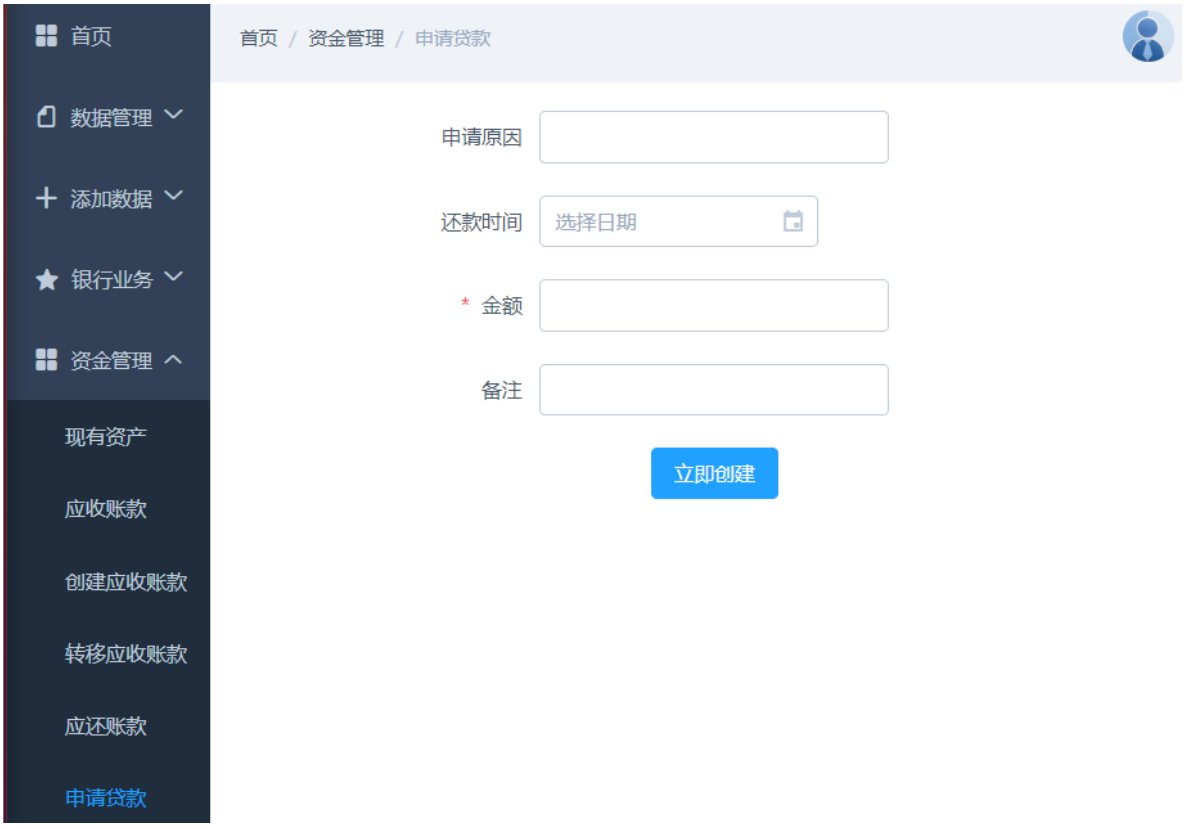
创建应收账款

这部分是核心企业通过银行认证的，拥有银行认证的token后可以将该token以应收账款的形式发放给交易的企业。注意到，这里由于是自己创建的应收账款，所以可以跟交易企业协商好还款时间等



申请贷款

在申请贷款的时候，企业写明申请原因，金额等信息，然后该贷款信息就会出现在银行的业务系统中



应还账款

企业发放应收账款给中下游企业后，中下游企业到期后可以向发放应收账款的企业申请偿还账款，如下所示。企业可以根据自己的情况，以及申请是否合规选择还款或者打回。

应收账款

收到应收账款的企业，可以根据付款时间等信息申请催还账款，催还信息会出现在对应企业的应还账款列表中。

待完善功能

1. 数据加密，数字签名，认证服务，零知识证明等密码学技术加强系统的隐私性
2. 更加严格的权限管理
3. 信用额度，信用评级，完善信用体系
4. 企业认证时的财产注册、抵押，银行回收债款时的拍卖等服务
5. 共识小组，允许部分企业之间成立小组，在组内实现更加严格完善的安全体系
6. 合约上共识机制的成立，使用博弈论的知识设计无第三方监管的交易合约
7. 添加仲裁机构的审查权限

心得体会

在本次作业中，我根据提供的供应链场景，基于FISCO-BCOS设计相关的智能合约并详细解释智能合约是如何解决提出的问题，并且将智能合约部署至链上，并调用相关函数，详细说明上述的四个功能具体是如何实现的。同时还构想了一下后期可以继续完善的功能。这只是大作业的第二阶段，在后面我将会不断完善合约的功能，并且实现相对应的前段界面等。