中山大学数据科学与计算机学院本科生实验报告

(2019 年秋季学期)

课程名称：区块链原理与技术

任课教师： 郑子彬

年级	2017 级	专业（方向）	软件工程
学号	17343117	姓名	吴荆璞
电话	18614050085	Email	1503493582@qq.com
开始日期	2019.11.12	完成日期	2019.12.13

一、 项目背景

在供应链金融领域，经常会出现需要延期支付的情况，而处在供应链下游且实力不那么强的公司常常需要用它们持有的大公司的拮据来进行融资，以证明它们具有偿还能力。这种信用评估机制不够灵活，大公司的信用只能直接传给与其有债务关系的企业，而无法继续向下传递。这给下游小公司的融资造成了很大的困难，且由于信息不透明也给银行等机构的风险评估带来了麻烦。

因此考虑一个联盟链的场景，银行等机构与众多企业作为节点，公司间的债务关系统一记录到区块链这个可信账本上，并且账本可拆分、转移，使得信用可在整个链上传递，降低企业的融资难度以及银行的评估成本。根据题目要求，实现应收账款上链、账款转让上链、利用账款向银行融资上链、账款结算上链四个基本功能。为此我设计了一套以银行为中心的信用评估机制，并增加了贷款自动发放、到期自动结算、违约惩罚等功能。

二、 方案设计

本次作业中借据等信息全部存储在链上（合约内变量），没有使用外部数据库。其中一些诸如银行账户余额等敏感信息为私有，拥有权限的用户可调用相关函数进行查询，而其余的则是 public 变量，任何用户均可可通过自带的与变量同名的函数进行查询。

主要部分的简略数据流图大致如下：

设计思路

根据我的理解，该合约最基本的功能是记账，本质上就是欠款方（债务人）给借款方（债权人）写一张欠条，发布到链上，以保证其公开、不可篡改，这样债权人就有了一个可靠的凭据，而债务人的公司的信用就可以顺着链上直接或间接持有其欠条的公司传递。

因此我首先定义一个欠条的数据结构，然后分别写两个函数：一个供债权人创建一张欠条并填入债务人、数额等关键信息，将其发布到链上；另一个供债务人对他的债权人发布的欠条信息进行确认。任何公司都可以调用该函数声称某人欠他任意数额的钱，但仅经过对方确认的欠条是有效的。确认表明债务人认可欠条的信息，承认自己欠欠条上写的债权人相应数额的钱，并对其上的信用担保、归还期限等信息予以认可。

下面是我定义的 receipt 数据结构：

```
struct receipt {
    uint rid;
    address Debtor;
    address Creditor;
    uint amount;
    bool confirmed;  //confirmed by the debtor
    bool evaluated;  //evaluated by the bank
    uint deadline;
    bool violated;  //true when the receipt is unpaid after the deadline
}
```

其中 rid 表示合约 id，Debtor 和 Creditor 分别表示债务人和债权人，amount 表示数额，confirmed 和 evaluated 分别表示是否经债权人确认，以及银行是否认可债务人的还款能力，后者可在拆分欠条时传递给新的欠条，以使信用可传递。deadline 表示截止期限，而 violated 则用于记录是否到期未还，它与后续的信用评价有关。

上述属性包括了欠条的一些基本信息，而至于相应金额对应的具体合约是什么，债务人用这些金额从债权人那里购买了什么产品或服务，暂不考虑，假定他们已经在线下协商好，而区块链只起一个账本的作用。

值得注意的是，这个联盟链并不是完全去中心化的，存在银行这个“超级节点”，它在各个公司的资产管理、信用评价、贷款发放等方面有着重要、不可替代的作用。

接下来要明确银行对债务人还款能力的评估部分。在一张欠条中，evaluated 属性为 true 表明银行认为该债务人有能力偿还相应数额的钱，银行的权威性使这张欠条被认为是低风险的（大概率能得到偿还）。这就将银行的风险评估与具体公司独立了出来，后续只需加入欠条的拆分转移功能，就可以将该信用传递下去。

我将该属性放在欠条中而非公司中，是因为不存在绝对的可信与不可信，而只有偿还能力的大小。将一个大公司设为可信，使它可以任意借钱，并认为所有欠条都是低风险的做法显然是错误的。反之一个小公司如果只借少量的钱，也可以被认为有偿还能力。因此评估不能简单地设置一个可信公司的列表，应结合具体欠条、具体金额和债务公司的还款能力考虑。

```
struct companyInfo {
    string companyName;
    string companyAddress;
    string description;
}

receipt[] public receipts;
address[] public companies;  //registered companies' list
mapping (address => int) balances;  //balances can't be public, function checkBalance allows company to check its own balance
mapping (address => int) public credits;  //credits are open to the public
mapping (address => companyInfo) public Infos;
```

如上图，除了存储所有欠条、所有公司信息、所有公司银行账户余额（不公开）的变量外，对每家公司维护一个“信用”变量，当它大于某张欠条上的数额时，就相信该公司有能力偿还。显然地，信用与公司在银行中的存款数有关。但除此之外，一些公司可能投入了很多在生产中，它的实际偿还能力可能远大于其在银行中资产数，对这种情况，银行可以对公司经营状况并调整其信用值。每家公司在银行都有一个 balance 和 credit，银行可分别通过 updateBalance 和 updateCredit 两个函数来设置这两个值：

```
function uodateBalance(address receiver, uint amount_, bool add) public returns (bool) {
    //can only call by the bank
    //add balance to a company
    require(msg.sender == 0xb820b9f01be068e273ac75e530f1f55c70cc648d);
    if (add == true) {
        balances[receiver] += amount_;
        credits[receiver] += amount_;  //if one owns more money, he has more credit
        return true;
    }
    else if (balances[receiver] >= amount_ && credits[receiver] >= amount_) {
        balances[receiver] -= amount_;
        credits[receiver] -= amount_;
        return true;
    }
    return false;
}
```

```
function updateCredit(address receiver, uint amount_, bool add) public returns (bool) {
    //can only call by the bank
    //add credit to a company
    require(msg.sender == 0xb820b9f01be068e273ac75e530f1f55c70cc648d);
    if (add == true) {
        credits[receiver] += amount_;
        return true;
    }
    else if (credits[receiver] >= amount_) {
        credits[receiver] -= amount_;
        return true;
    }
    return false;
}
```

上图中的 require 函数中的地址为银行的公钥地址，对所有用户可见，但含有 msg.sender 的函数在被调用时会有用户选项，银行用户可以用私钥对交易进行签名并通过公钥来验证它，因而其他用户即使知道上述公钥也无法冒充银行。

当存款数增加或减少时，信用额度自然等量地增加或减少，除此之外银行还可以根据评估上下调公司的信用。定义好信用后，就可以实现 makeReceipt 函数、confirmReceipt 两个函数来创建并发布欠条、确认欠条了：

```
function makeReceipt(uint rid_, address receiver, uint amount_, bool evaluated_, uint time) public returns (bool) {
    //call by the creditor
    //declare a receipt(the receiver owes me money)
    receipts.push(receipt({
        rid: rid_,
        Debtor: receiver,
        Creditor: msg.sender,
        amount: amount_,
        confirmed: false,
        evaluated: evaluated_,  //if true, means that the creditor require debtor's credit guaranty
        deadline: now + time,  //time indicates how many time left
        violated: false
    }));
    return true;
}
```

makeReceipt 函数由债权人调用，它创建一个欠条，填入 id、金额、债务人、是否要去信用担保、时间五个基本信息，并将其发布（创建一个实例并 push 到 receipts 数组中）。其中 confirmed 属性为 false，需要等待债务人确认。evaluated 属性代表该欠条中债务人是否被银行认可（有能力能够相应数额的钱），该属性是由银行根据债务人的情况确定的，债权人在这里的设置只表达了它对信用担保的要求，即若该属性为 false，债务人不需要银行的认可即可确认这张欠条，否则则需要有相应数额的信用做担保。而属性 deadline 是一个“时间戳”，债权人填入债务还有多久到期，函数自动计算到期的绝对时间。

```
function confirmReceipt(uint rid_, address sender_, uint amount_, bool evaluated_) public returns (bool) {
    //call by the debtor
    //confirm a receipt(I do owe the sender that amount of money)
    uint i;
    for (i = 0; i < receipts.length; i++) {
        if (receipts[i].Debtor == msg.sender && receipts[i].rid == rid_ && receipts[i].amount == amount_
        && receipts[i].Creditor == sender_ && receipts[i].evaluated == evaluated_) {
            //deadline is related to the function call, cannot be specified accuratedly
            if (receipts[i].evaluated == false) {  //don't need credit guaranty
                receipts[i].confirmed = true;
                return true;
            }
            //if credit guaranty is needed, the debtor can't confirm it unless it has enough credit
            else if (credits[receipts[i].Debtor] >= receipts[i].amount) {
                //the debtor's credit pass to the creditor with specified amount
                credits[receipts[i].Debtor] -= receipts[i].amount;
                credits[receipts[i].Creditor] += receipts[i].amount;
                receipts[i].confirmed = true;
                return true;
            }
        }
    }
    return false;
}
```

confirmReceipt 函数由债务人调用，它根据 id、债权人、金额、是否需要担保四个信息确认债务人为它的欠条，若没有条件完全匹配的欠条则失败，不会做任何事，否则将该欠条根据条件设置 confirmed 属性。若需要信用担保，则需要用债务人的信用做支付（余额不变，但信用减少 amount，因为信用中的这一部分被拿去为这张欠条做担保了）。这相当于将债务人信用的一部分传递给了债权人，债权人可利用它进行融资。此外，银行只需对某个公司做一次或定期的信用评估，评估结果就可被应用到对任意公司的任意数量的欠条上（只要总金额不超过信用额度），简化了信用评估流程。

需要注意的是，一次函数执行中的时间戳不是函数被调用时的时间，而是交易被打包的时间，因此这其中存在一些误差，不能严格地用相等条件判断，因此我在这里就没有将其放在条件中。债务人、债权人可通过下面两个函数查询与其相关的欠条，默认债务人在确认前已了解并认可时间信息（这两个函数返回对应欠条在 receipts 数组中的下标的集合，可据此调用公有变量自带的 receipts 查询对应欠条的具体信息）：

```
function checkReceiptDebtor(address company) public returns (uint[]) {
    require(company == msg.sender);
    //call by any company
    //check receipts on which he is the debtor
    uint[] ret;
    uint i;
    for (i = 0; i < receipts.length; i++) {
        if (company == receipts[i].Debtor && receipts[i].amount != 0) {
            ret.push(i);
        }
    }
    return ret;
}

function checkReceiptCreditor(address company) public returns (uint[]) {
    require(company == msg.sender);
    //call by any company
    //check receipts on which he is the creditor
    uint[] ret;
    uint i;
    for (i = 0; i < receipts.length; i++) {
        if (company == receipts[i].Creditor && receipts[i].amount != 0) {
            ret.push(i);
        }
    }
    return ret;
}
```

当然，如果一家公司作为债务人在一张欠条中出于信用暂时不足等原因暂时没能被 evaluated，而后面又有了足够的信用并愿意为该欠条担保，他可以通过 evaluateReceipt 函数向银行提出申请，若条件满足将使该欠条得到信任（但同时也需消费债务人的信用额度，在欠条创建时双方约定的是不需信用担保，因此这只能由债务人调用，债权人无权消费该信用来为他的欠条获得信任）：

```
function evaluateReceipt(uint rid_, address debtor_) public returns (bool) {
    uint i;
    //the bank believes that this debtor is able to pay this receipt
    for (i = 0; i < receipts.length; i++) {
        if (receipts[i].Debtor == debtor _&& receipts[i].rid == rid_ && receipts[i].evaluated == false
        && credits[receipts[i].Debtor] >= receipts[i].amount) {
            //the debtor's credit pass to the creditor with specified amount
            credits[receipts[i].Debtor] -= receipts[i].amount;
            credits[receipts[i].Creditor] += receipts[i].amount;
            receipts[i].evaluated = true;
            return true;
        }
    }
    return false;
}
```

有了基本的信用评估机制以及借条创建、确认后，下一步实现借条拆分转移并让信用评估属性得到继承看，以便在不需银行重复评估的情况下让信用方便地在整个供应链上传递。

当一个公司要向其他公司借钱时，它可以将自己持有的（作为债权人）的欠条拆分出来生成一张新的欠条给对方，如果原欠条是低风险的（经银行评估认可），那么新欠条会保持该属性，债务人也是原先的债务人。这样该公司就不用自己作为债务人了，因为那样的话它对欠条的偿还能力还有待认可，而债权人肯定更想要一份已经被认可的低风险的欠条。

我用一个 transferReceipt 函数实现上述功能。调用的公司传入 rid 参数，根据它的地址和 rid 找到它拥有的相应欠条（它必须是债权人且欠条已 confirm），这时根据 newid、newamount、newcreditor 三个参数创建相应 id、金额和债权人的新欠条。首先要比对新旧欠条的金额，仅前者不大于后者时可以做拆分。拆分时在旧欠条上减去相应金额，新欠条的债务人、evaluated 属性继承自旧欠条，而 confirmed 属性直接设为 true。区别于前面，这里是债务人而非债权人创建的欠条，他已经用自己拥有的欠条额度支付了该新欠条，因此应予以确认，没有人会通过声称自己欠别人多少钱并用自己的额度支付它而获利（区别于声称别人欠自己多少钱），即便出现他本来欠对方 300 万，却只拆出一张 200 万的欠条给对方的情况，链上也会有明确记录，对方可根据签订的协议与收到的欠条数额不等予以追讨：

```
function transferReceipt(uint rid_, uint newrid, uint newamount, address newcreditor) public returns (bool) {
    //call by company who owns a confirmed receipt and owes money to another company
    uint i;
    for (i = 0; i < receipts.length; i++) {
        //verify the ownership
        if (receipts[i].rid == rid_ && receipts[i].Creditor == msg.sender && receipts[i].confirmed == true) {
            //the receipt's owner wants to transfer part of it to a newcreditor
            if (receipts[i].amount < newamount)
                return false;  // can't afford
            else {
                receipts[i].amount -= newamount;

                receipts.push(receipt({
                rid: newrid,
                Debtor: receipts[i].Debtor,
                Creditor: newcreditor,
                amount: newamount,
                confirmed: true,  //has already paid(-amount), have to confirm here
                evaluated: receipts[i].evaluated,
                deadline: receipts[i].deadline,
                violated: false
            }));
            if(receipts[i].evaluated == true) {
                credits[receipts[i].Creditor] -= newamount;
                credits[newcreditor] += newamount;
            }
                return true;
            }
        }
    }
```

在上述函数中，如欠条是被银行认可的（evaluated 为 true），那么它上面的金额背后就有等量的信用担保，在拆分转移时信用也需要转移。

最后是欠条的偿还部分，债务人可以通过 repay 函数向债权人偿还欠条中规定数额的债务。值得注意的是，若欠条 evaluated 属性为 false，代表债务人没有做信用担保，因此在还款时就相当于一次普通的转账，信用、资金同步转移。而对已经担保过的欠条，债务人在当时虽然没有支付现金，但预支了信用，因此还款是只还钱而不再支付信用，这样才能保证一致性：

```
function repay(uint rid_, address creditor_, uint amount_) public returns (bool) {
    //call by the debtor
    //payback a receipt
    uint i;
    for (i = 0; i < receipts.length; i++) {
        if (receipts[i].Debtor == msg.sender && receipts[i].rid == rid_ &&
        receipts[i].Creditor == creditor_ && receipts[i].amount == amount_) {
            if (balances[msg.sender] < amount_)
                return false;  //can't afford
            else {
                receipts[i].amount = 0;
                balances[msg.sender] -= amount_;
                balances[receipts[i].Creditor] += amount_;
                if (receipts[i].evaluated == false) {
                    credits[msg.sender] -= amount_;  //haven't paid its credit, credit reduced here with the decrease of bala
                    credits[receipts[i].Creditor] += amount_;
                }

                return true;
            }
        }
    }
}
```

还完钱后不需要在 receipts 数组中找到对应位置的欠条并删除，更简单的做法是直接将金额会设成 0（如果还不起，将返回 false，不偿还可偿还的部分）。

除此之外，我还设置了一个 autoRepays 函数，由银行调用，遍历每家注册公司，对其作为债务人的且已到期的欠条，自动强制执行偿还，若金额不足以偿还，则判定为违约，违约信息会通过 violated 属性在欠条上记录，并扣除债务人 3 倍于欠条金额的信用值作为惩罚。

这里面欠条是一个静态的数据结构，其中的时间也是绝对的，它不能判断到期或在到期后自动执行任何动作，因此需要银行这个节点不断地进行轮询，每次调用该函数时根据当前时间做判断，并执行相应操作，因此这或许智能算一种“半自动”的到期偿还，不过我还没有想到有更好的办法，能够完全自动地实现上述功能：

```
function autoRepays() public returns (bool) {
    //can only call by the bank
    //check whether the companies have expired receipts to pay
    //If they have, try auto repay
    //If they don't have the money, note that they violated the contracts and reduce their credits
    require(msg.sender == 0xb820b9f01be068e273ac75e530f1f55c70cc648d);
    uint i;
    for (i = 0; i < receipts.length; i++) {
        if (now > receipts[i].deadline && receipts[i].amount != 0) {  //time is up but not repaid
            if (balances[receipts[i].Debtor] >= receipts[i].amount) {
                receipts[i].amount = 0;
                balances[msg.sender] -= receipts[i].amount;
                balances[receipts[i].Creditor] += receipts[i].amount;
                if (receipts[i].evaluated == false) {
                    credits[msg.sender] -= receipts[i].amount;  //haven't paid its credit, credit reduced here with the decre
                    credits[receipts[i].Creditor] += receipts[i].amount;
                }

            }
            else if (receipts[i].violated == false) {  //time is up but not repaid and can't afford
                receipts[i].violated = true;
                credits[receipts[i].Debtor] -= 3 * receipts[i].amount;  //punishment for violating the receipt
            }
        }
    }
    return true;
}
```

最后，有了上述比较完善的信用评价与转移的体系后，贷款申请就非常简单了：公司直接调用 loan 函数向银行提出需要的数额的贷款申请，银行根据其信用额度做判断，符合条件自动发放贷款，并扣除相应信用额度：

```
function loan(uint amount) public returns (bool) {
    uint i;
    if (credits[msg.sender] >= amount) {
        credits[msg.sender] -= amount;
        balances[msg.sender] += amount;
        return true;
    }
    return false;
}
```

其余诸如余额查询（不能作为公开变量，由一个带地址判断的函数实现）、公司注册、信息填写等无关主干的函数就不一一解释了，详细内容可见代码及注释。以上就是我的设计中的主要部分，它引入了比较完整的信用评价体系，以之为基础实现了应收账款上链、应收账款拆分转让、利用应收账款向银行融资以及支付结算功能，并借助银行这个

“超级节点”实现了资金、信用管理和到期自动偿付。合约的源代码在 contract 目录下的 test.sol 和 test.txt 中（两者内容一样）。

三、 功能测试

对于上述合约，我设计了一个尽可能精简的测例，它测试了合约中所有主要函数。测例包含银行、公司 A、公司 B、公司 C 四个角色，它们的公钥地址如下：

```
Bank: 0xb820b9f01be068e273ac75e530f1f55c70cc648d

Company_A: 0x43c4d4fe096740dfa1c7ebd186246dc785e238d3

Company_B: 0xc75698618c5f890f699d92a4eecb750cc7abf951

Company_C: 0xe18b3352f2cd1cf73caacadc742904e5e9bea3a3
```

正常情况下他们应该是 4 个独立的节点，但为测试方便我采用了单节点多用户的模式，测试使用了 Webase 区块链浏览器，测试主要流程如下：

1.银行部署 test008 合约

2.银行通过 updateBalances 函数给 A 公司账户增加 50000 余额，通过 updateCredits 函数给 A 公司增加 80000 信用

3.B 公司通过 makeReceipt 函数向 A 公司发起 id 为 101 金额为 10000 的欠条，要求信用担保，且时间足够长

4.A 公司通过 confirmReceipt 函数确认该欠条，并用其 10000 的信用做担保

5.B 公司通过 transferReceipt 函数将欠条金额中的 3000 拆给 C 公司，新欠条 id 为 201

6.C 公司通过 loan 函数向银行贷款 2000

7.A 公司调用 repay 函数，偿还对公司 C 的 id 为 201 的欠条

8.C 公司通过 makeReceipt 函数向 B 公司发起 id 为 101 金额为 2000 的欠条，要求信用担保，且时间比较短

9.B 公司通过 confirmReceipt 函数确认该欠条，并用其 2000 的信用做担保

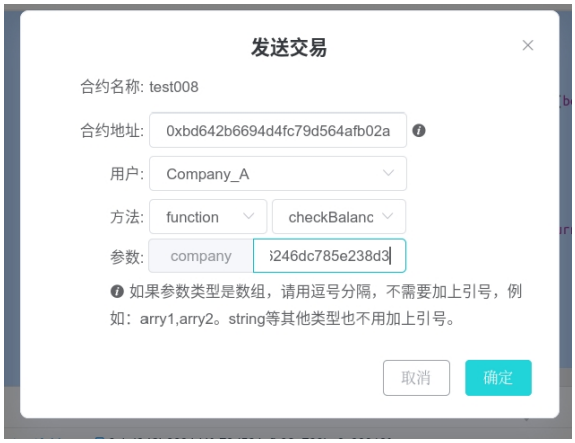
10.到期后，银行调用 autorepay 函数自行自动偿付，B 公司余额不足，记录违约并做出相应惩罚，扣除 6000 信用，剩余 1000

首先银行调用 updateBalance 函数向公司 A 账户打 50000 元：

返回结果显示函数成功执行，返回 true：

类似地，调用 updateCredit 函数额外给予 A 公司 80000 的信用：

调用 checkBalance 函数和 credits 函数（共有变量自带的查询函数）查询 A 公司的余额和信用，结果为 50000 和 130000（50000+80000），说明第一步正确：



第二步，公司 B 向公司 A 发起欠条，金额为 10000，要求信用担保，为测试方便截止时间设置的很长，暂时不会到期：

这是第一个交易，查询显示尚未确认：

A 公司调用 confirmReceipt 函数确认交易，并用自己的信用额度为该欠条做担保：

再次查询显示确认成功（第 5 行的 confirm 属性变为 true）：

接下来公司 B 调用 transferReceipt 函数将它拥有的已确认且信用被认可的欠条拆出 3000 给公司 C：

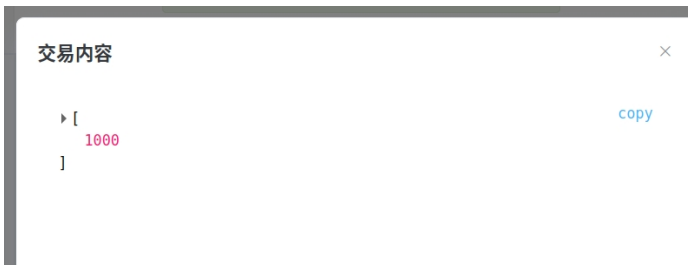
查询新欠条结果：

此时分别查询三家公司的信用，公司 A 在为第一张欠条做担保时，转移了 10000 的信用额度给公司 B，公司 B 又将欠条的一部分转给公司 C，信用跟着转移，最终公司 A 信用 130000-10000=120000，公司 B 信用 10000-3000=7000，公司 C 信用 0+3000=3000（如下三图所示）：

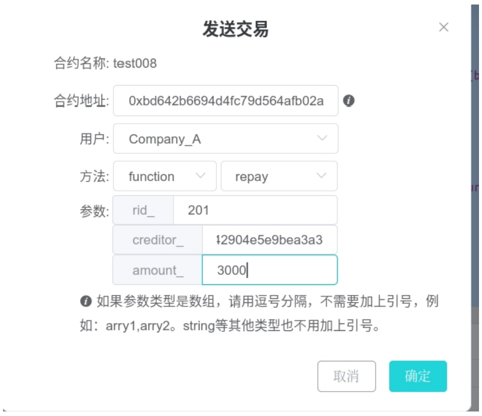
公司 C 可用得到的 3000 信用做担保，调用 loan 函数向银行贷款 2000，银行根据其信用值自动做条件判断，发放贷款，扣除相应额度的信用：

此时再查询公司 C 余额，显示 2000 元已到账：

而公司 C 的信用消耗也为 2000，信用变为 1000：

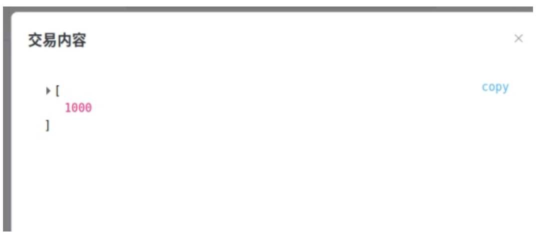


接下来公司 A 调用 repay 函数偿还对三的金额为 3000 的贷款：

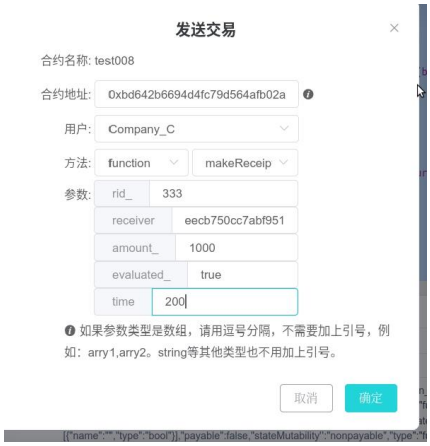


偿还后公司 C 得到 3000 元，余额变为 5000：

但它的信用还是 1000，因为这张欠条时用信用担保的，在确认欠条时已进行了信用的转移，而资金尚未转移，这里只是增加余额。若是未担保的欠条，确认欠条是对双方信用无影响，归还时则相当于一笔转账，金额和信用同步转移，都要增减：

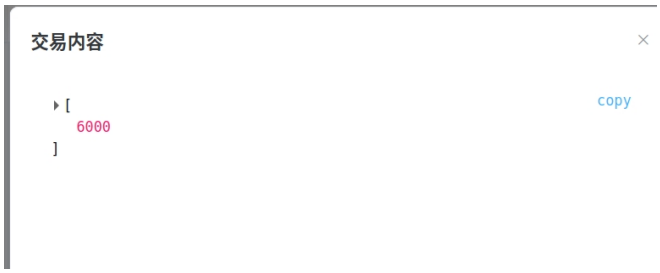


这时假设公司 C 向公司 B 发起了一个需要担保的额度为 1000 的欠条，为便于测试时间设置的很短：

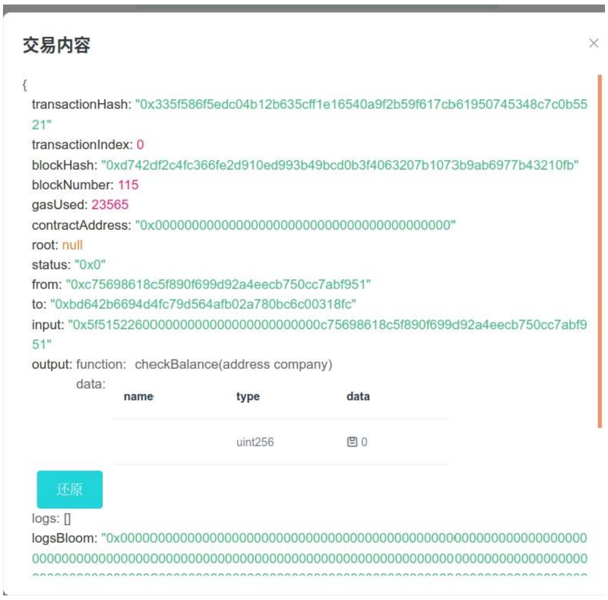


公司 B 进行确认：

公司 B 为该欠条做信用担保消耗其 1000 的信用额度，剩余 6000:



目前公司 B 持有的被认可的资产只有拆分剩下的一的数额为 7000 的欠条，它并没有现金：



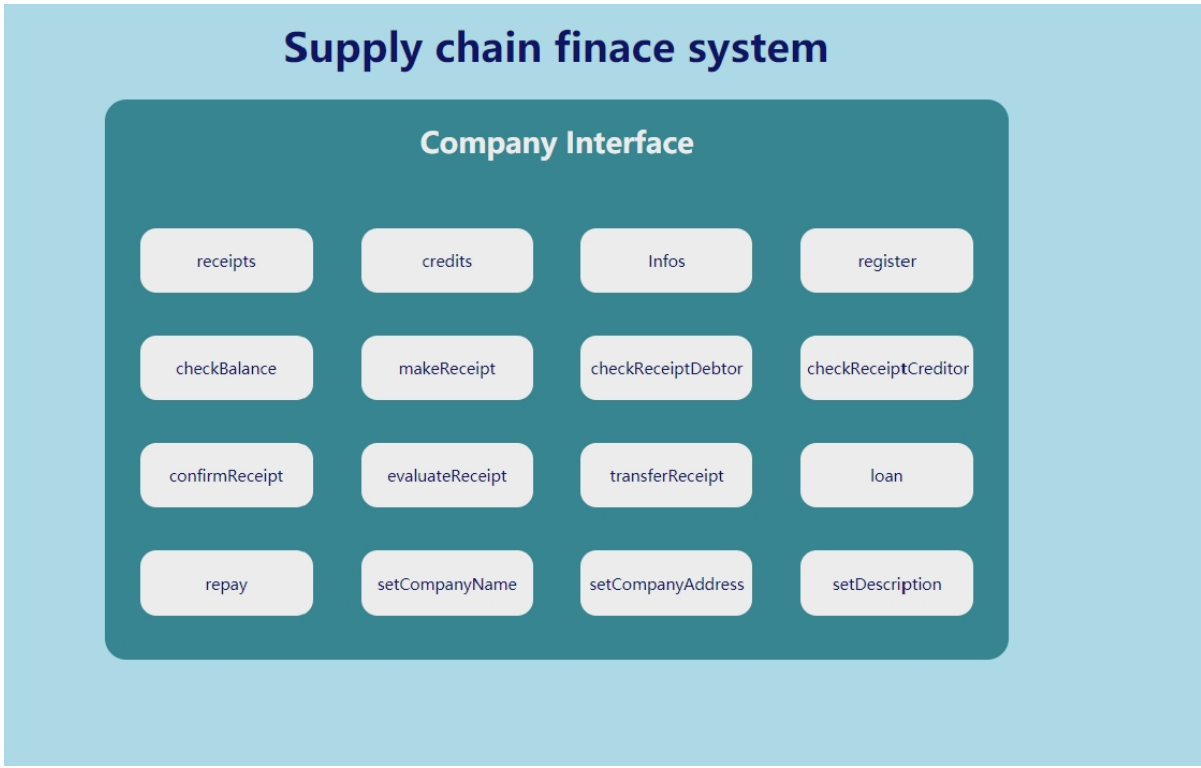
因此在公司 A 还它钱之前，它实际上并不具有任何偿还能力，由于时间设置的很短，它对 C 的欠条很快到期，假设公司 A 还没有还它钱，而银行调用函数进行到期欠条自动结算，公司 B 账户里没有钱，发生违约，该欠条的违约属性变为 true，二被扣除 3 倍于违约欠条金额的信用（3*1000=3000）作为惩罚，信用剩下 3000：

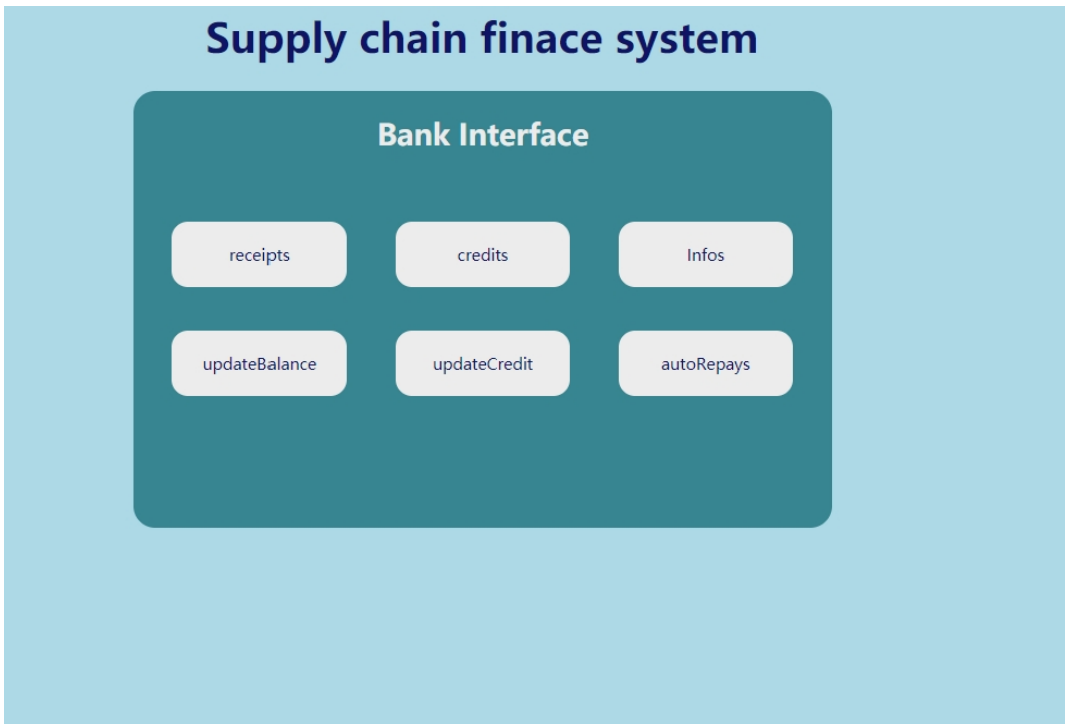
至此完成了基本功能测试，涉及合约中 updateBalance、updateCredit、chackBalance、makeReceipt、confirmReceipt、evaluateReceipt、transferReceipt、loan、repay、autoRepays 等函数，这些函数功能符合预期。

四、 界面展示

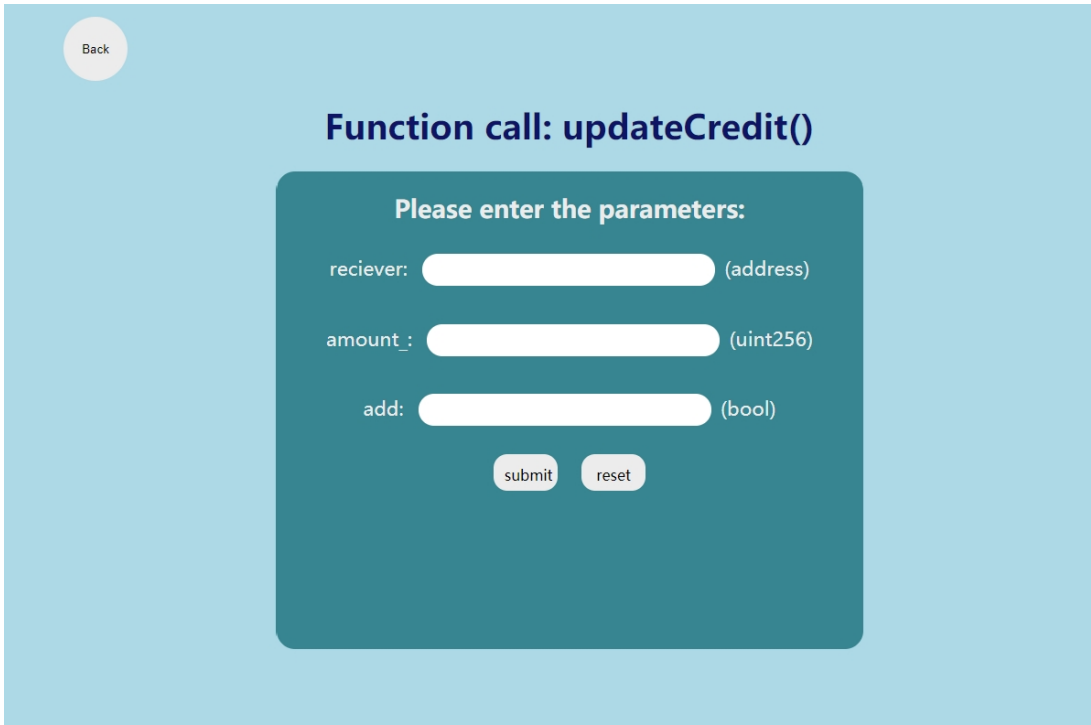
接下来的任务是设计一个前端界面，然后与服务端连接，后者通过 sdk 提供的接口与区块链上的数据进行交互，执行函数调用并返回结果。

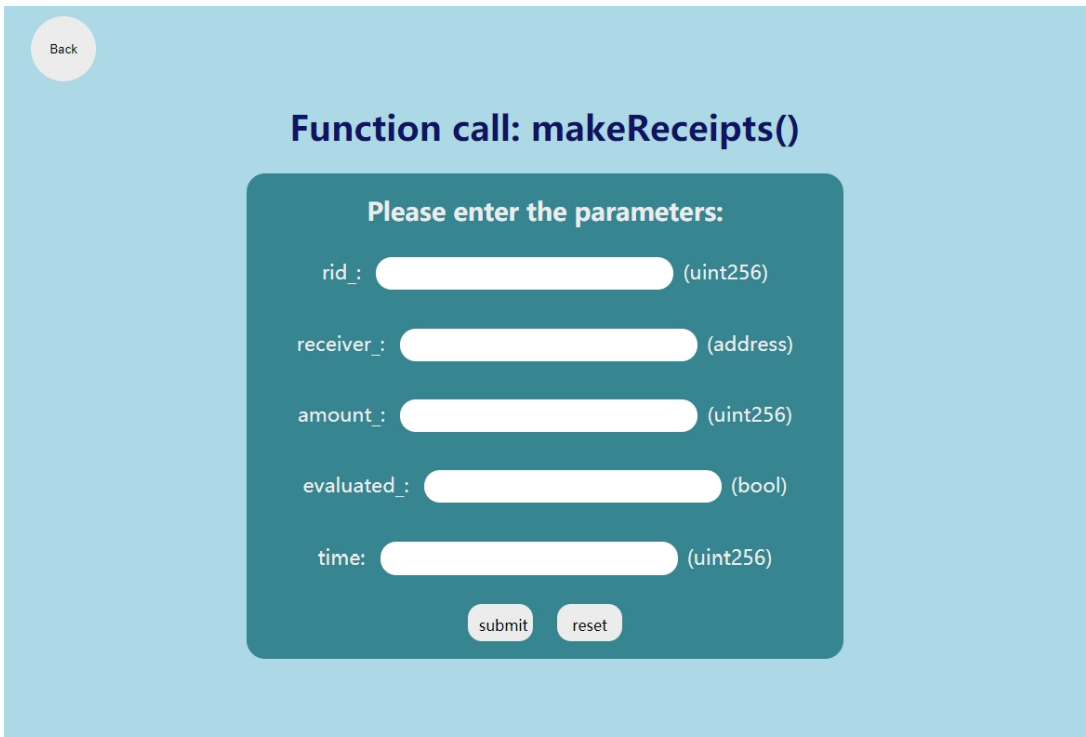
首先是界面设计，分为公司和银行两种，它们所能调用的函数有所不同，每个函数用一个按钮表示，如下两图所示：





点击按钮会进入相应函数的调用界面，该界面包含一张表单和一个返回按钮，表单中列出了该函数需要的参数名称和参数类型，填入并提交后 post 出去交由服务端处理。以银行调用的 updateCredit 函数和公司调用的 makeReceipts 函数为例，界面如下两图所示：





详细内容可见 Front 目录中的 company.html 和 bank.html。

接下来是后端的工作，首先要做到通过 sdk 与链上节点建立连接。我使用的是 FISCO BCOS 提供的 java sdk，按照文档提示安装 web3sdk 并完成相关配置后，进一步安装 spring-boot-starter 并进行相关配置。我本来想使用的是之前搭建的单群组 4 节点的联盟链，rpc 端口为 20000 到 20003，据此修改 application.yml 中的相关内容。但后来发现即便我完成了依赖添加、证书拷贝等操作后，仍然无法通过 spring-boot-starter 中的 test，观察报错信息后发现 solidity 到 java 编译功能无误，但无法正常调用 API（getBlocknumber 等）或部署智能合约（Helloworld）。后来发现它好像只能连接一个节点，在删除后面 3 个节点的端口号，仅保留 20000 后即可通过 test。按照 GitHub 上文档的说明，该测试包含 solidity 到 java 编译、部署合约、使用 API 等功能的测试，表明已成功建立与区块链节点的连接，并可部署合约、通过 API 获取区块链上的数据：

```
encrypt-type: # 0:standard, 1:guomi
  encrypt-type: 0

group-channel-connections-config:
  caCert: classpath:ca.crt
  sslCert: classpath:sdk.crt
  sslKey: classpath:sdk.key
  all-channel-connections:
    - group-id: 1 #group ID
      connections-str:
        - 127.0.0.1:20200 # node listen_ip:channel_listen_port

channel-service:
  group-id: 1 # The specified group to which the SDK connects
  agency-name: fisco # agency name

accounts:
  pem-file: 0xcdcce60801c0a2e6bb534322c32ae528b9dec8d2.pem
  p12-file: 0x98333491efac02f8ce109b0c499074d47e7779a6.p12
  password: 123456
```

```
fisco-bcos@fiscobcos-VirtualBox:~/fisco$ cd ../
fisco-bcos@fiscobcos-VirtualBox:~$ cd spring-boot-starter
fisco-bcos@fiscobcos-VirtualBox:~/spring-boot-starter$ ls
build          CONTRIBUTING.md  gradlew      README.md         src
build.gradle   doc              gradlew.bat  release_note.txt
Changelog.md   gradle           logs         settings.gradle
fisco-bcos@fiscobcos-VirtualBox:~/spring-boot-starter$ ./gradlew build
Starting a Gradle Daemon, 1 incompatible and 1 stopped Daemons could not be reused, use --status for details

Deprecated Gradle features were used in this build, making it incompatible with Gradle 6.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/5.6.2/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 1m 6s
3 actionable tasks: 3 executed
fisco-bcos@fiscobcos-VirtualBox:~/spring-boot-starter$ ./gradlew test

Deprecated Gradle features were used in this build, making it incompatible with Gradle 6.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/5.6.2/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 52s
5 actionable tasks: 2 executed, 3 up-to-date
fisco-bcos@fiscobcos-VirtualBox:~/spring-boot-starter$
```

```
fisco-bcos@fiscobcos-VirtualBox:~/fisco/console$ ./sol2java.sh org.fisco.bcos.test.contract

Compile solidity contract files to java contract files successfully!
fisco-bcos@fiscobcos-VirtualBox:~/fisco/console$
```

后面我编译了 test 合约，参照文档中的示例尝试部署我的 test 合约并调用一个简单的 credits 函数，不过这一部分比较复杂，我先后遇到了很多问题，目前还没有跑通，因此就不在此处展示该部分代码了。该项目放在 Server 目录下。

五、 心得体会

区块链的核心是解决信用问题，而涉及到大量资金流动金融领域正是对信用问题最敏感的，因此区块链天然地可以在这方面发挥作用。作为一个公开，透明的系统，谁可以调用哪些函数被定义的很清楚，而诸如欠条等账本也以变量的形式存储在链上，除合约中定义的合法方式外不可篡改。这样就可以形成一个各方公认的可信记账平台，解决由信息不对称带来的高昂的信用评估成本。

本次大作业提供了一个比较开放的供应链金融场景，对四个基本功能，可以自己定义各方的角色、权限以及实现方式。通过这次合约的设计，我对区块链以及智能合约的应用有了更具体的理解。在我的设计中，银行这个超级节点在资金、信用管理以及到期自动结算等方面发挥了重要作用。虽然区块链的一个重要特征是去中心化，但在很多金融方面的应用领域，常常存在银行等权威机构，这些机构往往利用其在现实世界中的权威和信用来管理一些普通节点说了不算的事情，比如信用评估等。介于完全去中心化的共有链和近似于一个普通的数据库的私有链之间的联盟链渐渐成为企业间协作中最受欢迎的平台。

对我来说，这次大作业中最困难的不是链上的部分，而是如何用 sdk 串通后端、链端、前端。我之前没有学习过相关的内容，对用到的工具也不是很熟悉，因此在文档没有提及或按文档操作遇到错误是常常不知道怎么处理。但这方面的技术还是比较重要的，日后我还需要多多学习。