《区块链原理》课程设计报告

题目：	基于区块链的供应链金融平台		
小组成员信息			
姓名	**学号**	**班级**	**分工**
甘家振	18340043	计科2班	前端/后端/链端
胡邱诗雨	18340056	计科3班	前端/链端
谢善睿	18340184	计科7班	前端/链端

演示地址：http://bc.jxzhn.cn:45678，临时有效。

一 链端需求分析及接口

1.1 需求分析

本项目基于区块链技术，要求实现供应链中交易信息透明化，以及实现核心企业的信用随着供应链的流向传递。通过引入供应链外的可信机构在区块链上确认供应链的交易，可以保证交易信息在区块链上透明化。利用区块链的不可篡改性，保证在区块链上记录的信息的真实性，可以解决信用不能传递引出的问题。

根据上述需求，我们需要在智能合约中维护两个结构体：

```
struct Company {
    string name;
    uint amount;
    uint field; // 使用0表示公司为其他领域，1表示公司为金融机构
    bool registered; // mapping里面判断是否已存在
}
struct Receipt {
    address from;
    address to;
    uint amount;
    uint status; // 使用0表示账单未清算，1表示已清算
}
```

用于记录区块链上的公司信息和账单信息。拥有上述数据结构的智能合约能够记录在供应链上的不同公司之间，使用核心企业信用作为担保的应收账款单据（下称账单）的交易账单，从而解决信用不能沿供应链传递的问题。

1.2 智能合约接口

我们的合约提供接口，使每一个加入合约的公司都能够通过自己的私有地址以及公司信息，映射到自己在合约中的 Company 数据结构。同时在每一家新加入合约的公司登记时，会获得一个身份：一般公司、核心企业、金融机构、认证机构四者中的一个。这一个登记过程通过函数接口

```
// 注册一个公司，只能由认证机构调用
// 返回值：0表示成功，-1表示已存在
//         -2表示该公司为金融机构无法为核心企业
//         -3表示核心企业已存在
function registerCompany(address addr, string name, uint amount, uint field, uint isKernel) public returns(int)
```

来实现。我们在合约中仅赋予认证机构登记公司信息。

我们通过函数接口

```
// 功能一：实现采购商品—签发应收账款交易上链
// 返回值：0表示成功
function createReceipt(address to, uint amount) public returns(int)
```

实现采购商品--签发应收账款交易上链的功能。此函数实现了允许核心企业使用账单来完成交易，并且函数把交易过程写入区块链以保证其透明性和可靠性。

我们通过函数接口

```
// 功能二：实现应收账款的转让上链
// 返回值：0表示成功，-1表示可转让账单金额不足
function transferReceipt(address to, uint amount) public returns(int)
```

实现应收账款的转让上链的功能。此函数实现了某个注册公司使用自己已有的账单向别的公司采购时的支付过程。同时本函数维护了在使用账单时，区块链上信息的准确性。

我们通过函数接口

```
// 功能三：利用应收账款向银行融资上链
// 返回值：0表示成功，-1表示to参数不是金融机构，
//         -2表示金融机构账户额度不足，
//         -3表示请求融资的机构账单额度不足
function useReceipt(address to, uint amount) public returns(int8)
```

实现利用应收账款向银行融资上链的功能。此函数实现了注册公司使用账单向金融机构申请融资时，区块链做出的响应变化。此函数保证了注册公司在融资时的便捷性（体现在省去了传统供应链上验证账单合法性的步骤）。

我们通过函数接口

```
// 功能四：应收账款支付结算上链
// 返回值：0表示成功，-1表示金额不足,清算失败
function settleReceipt() public returns(int)
```

实现应收账款支付结算上链的功能。这个函数解决了核心企业有足够多的资金，想要清算自己在供应链上的欠款额度时的需求。

二 后端接口实现

我们使用 Node.js 作为后端。通过使用 FISCO-BCOS 官方提供的 Node.js SDK (https://github.com/FISCO-BCOS/nodejs-sdk)，我们可以方便地在 Node.js 中完成对区块链上智能合约的编译、部署及调用。在这里，我们使用 Node.js 完成对 solidity 编写的合约的编译和部署，而不选择通过控制台部署，这是为了更好的管理合约的 ABI 和地址信息，方便在后端中使用。

需要注意的是，由于 FISCO-BCOS 官方提供的 Node.js SDK 需要在配置文件中预先写好所有区块链账户私钥信息，出于实现方便考虑，我们在演示项目中直接使用了一个服务器管理所有的私钥，而不是动态的进行类似于登录的操作。实际上，私钥应当由各个公司独自管理。

2.1 合约编译

合约编译功能实现在 `compile.js` 中。核心代码如下：

```
let compServ = new CompileService(config);

let name = await input('Which contract are you gonna compile? ');

console.log('Compiling ...');
let contractClass = compServ.compile(`contracts/${name}.sol`);

console.log('Compilation finished.');
fs.writeFile(`compiled/${name}.json`, JSON.stringify(contractClass), (err) => {
    if (err) {
        console.log('Failed to wirte compiled file.');
        console.log(err);
    } else {
        console.log('Compiled file saved.');
    }
});
```

我们将合约编译后，存储合约的 ABI 信息和二进制代码（以十六进制字符串格式提供）到一个 JSON 文件中。这些内容将会在合约部署与调用中用到。

2.2 合约部署

完成编译后，通过 `deploy.js` 将合约部署到链上。核心代码如下：

```
let name = await input('Which contract are you gonna deploy? ');

console.log('Loading contract from compiled file ...');
let compiled = JSON.parse(fs.readFileSync(`compiled/${name}.json`))

let account = await input('Which account are you gonna use? ');
let parameters = (await input('Parameters of constructor (split by space):
')).split(
    /\s+/).filter(item => Boolean(item)); // 去除空串

console.log(`Trying to deploy contract ${name} ...`);
try {
    let res = await web3j.deploy(compiled.abi, compiled.bin, parameters,
account);
    console.log(res);

    fs.writeFile(`deployed/${name}.json`, JSON.stringify(res), (err) => {
        if (err) {
            console.log('Contract deployed, but failed to write deployed file.');
            console.log(err);
        } else {
            console.log('Deployed file saved.');
        }
    });
} catch(err) {
    console.log('Failed.');
    console.log(err);
}
```

返回值是一个 JS 对象，其中包括有合约的地址信息，该地址需要在调用合约时指定。我们将该对象保存到 JSON 文件中，以在后端中读取地址。

2.3 提供后端 HTTP 服务，完成合约调用

由于前端代码无法直接与区块链进行交互，我们使用 Node.js 后端来完成合约调用的工作，而前端通过 HTTP 请求与后端进行通信。后端实现位于 `app.js` 文件中。

首先，我们从合约编译和部署时保存的 JSON 文件中读取合约的 ABI 和地址信息，构造一个 FISCO-BCOS Node.js SDK 定义的合约实例对象。

```
console.log('Loading contract from compiled file ...');
let compiled = JSON.parse(fs.readFileSync('compiled/SupplyChain.json'))

let contract = createContractClass(
    compiled.name, compiled.abi, compiled.bin, config.encryptType
).newInstance();

console.log('Loading deployed contract address from deployed file ...');
let contractAddr = JSON.parse(fs.readFileSync('deployed/SupplyChain.json'))
['contractAddress'];
contract.$load(web3j, contractAddr);

console.log('Done.');
```

我们使用 express 库提供 web 前端的 HTTP 站点及一个 HTTP 合约调用服务。

```
// 准备HTTP服务
const express = require('express');
const bodyParser = require('body-parser');
const port = 45678;

var app = express();
app.use(bodyParser.json({ limit: '5mb'}));

app.use(express.static('web', {index: '/test.html'})); // 网页根目录

// 接口参数如下
// account: 字符串，调用合约的账户名，必须是config.json中已有的账户
// method: 字符串，想要调用的合约方法名字
// parameters: 列表，合约方法调用参数
// 返回一个JSON对象字符串
// ok: 布尔值，调用是否成功
// msg: 字符串，如果ok为true，则设为'OK'，否则为错误信息
// data: 列表，合约方法调用的返回值
app.all('/contractMethod', async (req, res) => {
    let reqData = getReqData(req);
    console.log(`call 'contractMethod' from ip ${req.ip}, params:
${JSON.stringify(reqData)}`);

    if (typeof(reqData.account) != 'string' || typeof(reqData.method) != 'string'
||
        !Array.isArray(reqData.parameters)
    ) { // 检查接口参数类型
        console.log('failed at parameter type checking.');
        res.json({ok: false, msg: 'Bad iterface call.', data: []});
        return;
    }
```

```
30
31      // 进行合约方法调用
32      try {
33          contract.$by(reqData.account);
34          let retval = await contract[reqData.method](...reqData.parameters);
35          console.log(`retval: ${JSON.stringify(retval)}`);
36          res.json({ok: true, msg: 'OK', data: retval});
37      } catch (err) { // 出错
38          let errString = err.toString();
39          console.log(errString);
40          res.json({ok: false, msg: errString, data: []});
41      }
42  });
43
44  var server = app.listen(port);
45  console.log(`server started at port ${port}.`)
```

三 前端实现

使用了 Vue 作为前端用户界面的框架，以及 Bootstrap4 的前端组件库，用来搭建前端。

3.1 HTML文件调用智能合约方法

- 函数 `contractMethod` 调用智能合约中的函数，参数为 `account`：当前用户的名字，`methos`：调用的智能合约函数名字以及 `parameters`：需要传递给智能合约函数的参数。然后从后端返回智能合约的返回值，即可判断是否成功执行。

```
1   async function contractMethod(account, method, parameters) {
2           try {
3               let res = await axios.post('/contractMethod', {
4                   account: account,
5                   method: method,
6                   parameters: parameters,
7               });
8               if (res.data && res.data.ok) {
9                   return res.data.data;
10              } else {
11                  console.log(JSON.stringify(res.data? res.data.msg: res));
12              }
13          } catch (err) {
14              console.log(`error occurred when calling contract method ${method}
    with ${parameters} by ${account}.`);
15              console.log(err);
16          }
17      }
```

3.2 HTML文件数据维护

- vue的基本框架

  其中因为智能合约的参数是直接利用地址对应各个账户，但是对于用户来说这是不够直观的，因此存储了一个字典用于将地址和名称对应，方便用户直接选择公司名称，然后根据字典查询出地址，传递给智能合约。

```
1   var appVue = new Vue({
2       el: '#app',
3       data: {
4           accounts: [],
```

```
        user: 'car',
        currentAccount: {},
        buyFrom: '',
        buyAmount: 0,
        transferTo: '',
        toAmount: 0,
        useTo: '',
        useAmount: 0,
        // 字典
        nameToAdd: {'car': '0x19b267f1c7a491ea721b155d51fd447c50bed160',
                    'bank': '0x38de065475130a8629ee502ef732c9547f67f713',
                    'tyre': '0x7bced38f4dc58648417b9784e65dcc8564cf899b',
                    'hub': '0xa75ac687c81088b567ea399679efa29ce4991907'},
        addToName: {'0x19b267f1c7a491ea721b155d51fd447c50bed160': 'car',
                    '0x38de065475130a8629ee502ef732c9547f67f713': 'bank',
                    '0x7bced38f4dc58648417b9784e65dcc8564cf899b': 'tyre',
                    '0xa75ac687c81088b567ea399679efa29ce4991907': 'hub'},
    },
    method:{
    ...
    },
})
```

- 根据当前用户的页面需要展现的数据，进行数据的更新。主要调用智能合约中的查询函数，返回当前所有用户最新的 `receipts` 以及 `debt`， `amount`。

```
async refresh() {// 更新数据
    this.accounts = [];
    for (let name of ['car', 'bank', 'tyre', 'hub']) {// 重新取一下数据
        let account = {name: name};
        // 查询余额
        try {
            let [status, amount] = await contractMethod(name, 'getAmount', []);
            account.amount = amount;
        } catch (err) {
            console.log(err);
        }
        // 查询debt
        try {
            let [debt, indice] = await contractMethod(name, 'getDebts', []);
            account.debt = debt;
            account.debtList = await getReceiptsDetail(name, indice);
        } catch (err) {
            console.log(err);
        }
        // 查询receipt amount
        try {
            let [receiptAmount, indice] = await contractMethod(name, 'getReceipts', []);
            account.receiptAmount = receiptAmount;
            account.receiptList = await getReceiptsDetail(name, indice);
        } catch (err) {
            console.log(err);
        }
        this.accounts.push(account);
    }
},
```

- 切换用户时更改user来确保各项数据一致更新，并且将当前页面需要展示的数据更新。

```
changeUser(newUser) {
    this.user = newUser;
    for(let idx = 0; idx < this.accounts.length; idx++) {
        if(this.accounts[idx].name == this.user){
            this.currentAccount = this.accounts[idx];
            break;
        }
    }
    this.buyFrom = '';
    this.buyAmount = 0;
    this.transferTo = '';
    this.toAmount = 0;
    this.useTo = '';
    this.useAmount = 0;
},
```

- 其中获取receipts和debt详情的函数为getReceiptDetail函数，根据参数用户的姓名以及单据的长度，返回每一份单据的详情。

```
async function getReceiptsDetail(account, indice) {
    let receiptList = [];
    for (let i of indice) {
        let res = await contractMethod(account, 'getReceiptDetail', [i]);
        receiptList.push(res[0]);
    }
    return receiptList;
}
```

- 效果展示：

- 功能一：实现采购商品--签发应收账款交易上链。首先调用智能合约`createReceipt`，然后根据合约的返回值判断是否交易成功，若交易成功则调用`refresh`函数以及`changeUser`函数对页面展示的数据进行更新。若交易不成功则弹出弹窗提示失败的原因。并且每次都要将与html中的`input`组件利用`v-model`双向绑定的变量初始化。

```
async buy() {//生成账单
    if(this.buyFrom == "") {
        alert("收款企业不能为空！");
        console.log("Fail!");
    }
    else {
        let result = await contractMethod(this.user, 'createReceipt',
[this.nameToAdd[this.buyFrom], this.buyAmount]);
        if(result == 0)
        {
            alert("账单生成成功！")
            console.log("success!");
            await this.refresh();
            this.changeUser(this.user);
        }else if(result == -1)
        {
            alert("公司未注册，生成账单失败！")
            console.log("Fail!");
        }else if(result == -2)
        {
            alert("当前公司不是核心企业，无法发放账单！")
            console.log("Fail!");
        }
    }
    //置0
    this.buyFrom = '';
    this.buyAmount = 0;
},
```

  - 效果展示

    选择企业，输入金额后，点击确认按钮即可

- 功能二：实现应收账款的转让上链。首先调用智能合约`transferReceipt`，然后根据合约的返回值判断是否转让成功，若转让成功则调用`refresh`函数以及`changeUser`函数对页面展示的数据进行更新。若转让不成功则弹出弹窗提示失败的原因。并且每次都要将与html中的`input`组件利用`v-model`双向绑定的变量初始化。

```
async transfer() {//拆分转移账单
    if(this.transferTo == "") {
        alert("转让目标不能为空！");
```

```
            console.log("Fail!");
        }
        else {
            let result = await contractMethod(this.user, 'transferReceipt',
    [this.nameToAdd[this.transferTo], this.toAmount]);
            if(result == 0)
            {
                alert("账单转让成功！")
                console.log("success!");
                await this.refresh();
                this.changeUser(this.user);
            }else if(result == -1)
            {
                alert("公司未注册，转让失败！")
                console.log("Fail!");
            }else if(result == -2)
            {
                alert("可转让金额不足，转让失败！")
                console.log("Fail!");
            }
            this.transferTo = '';
            this.buyAmount = 0;
        }
    },
```

- 效果展示

  选择企业，输入金额后，点击确认按钮即可

- 功能三：实现利用应收账款向银行融资上链。首先调用智能合约`useReceipt`，然后根据合约的返回值判断是否融资成功，若融资成功则调用`refresh`函数以及`changeUser`函数对页面展示的数据进行更新。若融资不成功则弹出弹窗提示失败的原因。并且每次都要将与html中的`input`组件利用`v-model`双向绑定的变量初始化。

```
    async useReceipt() {//融资
        if(this.useTo == "") {
            alert("目标机构不能为空！");
            console.log("Fail!");
        }
        else {
            let result = await contractMethod(this.user, 'useReceipt',
    [this.nameToAdd[this.useTo], this.useAmount]);
            if(result == 0)
            {
                alert("融资成功！")
                console.log("success!");
```

```
            await this.refresh();
            this.changeUser(this.user);
        }else if(result == -1)
        {
            alert("公司未注册，融资失败！")
            console.log("Fail!");
        }else if(result == -2)
        {
            alert("融资对象不是金融机构，融资失败！")
            console.log("Fail!");
        }else if(result == -3)
        {
            alert("金融机构账户额度不足，融资失败！")
            console.log("Fail!");
        }else if(result == -4)
        {
            alert("融资支票额度不足，融资失败！")
            console.log("Fail!");
        }
        this.useTo = '';
        this.useAmount = 0;
    }
},
```

- 效果展示

  选择企业，输入金额后，点击确认按钮即可

- 功能四：实现应收账款支付结算上链。首先调用智能合约`settleReceipt`，然后根据合约的返回值判断是否清算成功，若清算成功则调用`refresh`函数以及`changeUser`函数对页面展示的数据进行更新。若清算不成功则弹出弹窗提示失败的原因。

```
async clearDebt() {//清算
    let result = await contractMethod(this.user, 'settleReceipt', []);
    if(result == 0)
    {
        alert("清算成功！")
        console.log("success!");
        await this.refresh();
        this.changeUser(this.user);
    }
    else if(result == -1)
    {
        alert("当前企业不符合清算条件，清算失败！")
        console.log("Fail!");
    }else if(result == -2)
    {
```

```
16            alert("当前企业余额不足以清算，清算失败！")
17            console.log("Fail!");
18        }
19    },
```

- 效果展示