

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

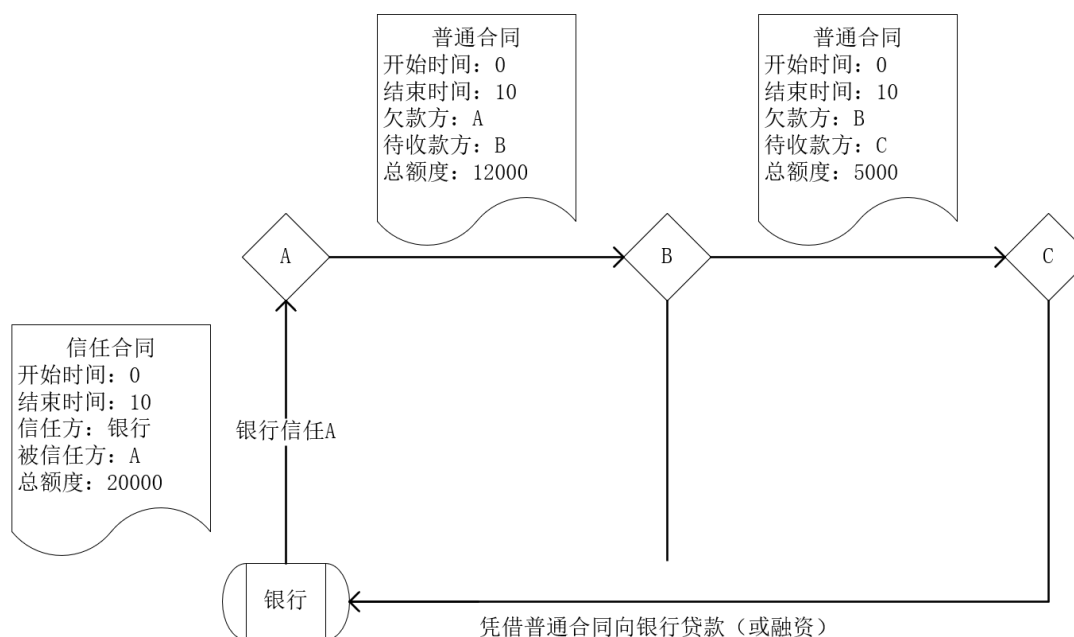
任课教师：郑子彬

年级	17 级	专业（方向）	软件工程
学号	17343082	姓名	陆宇霄
电话	15013027301	Email	luyuxiaorc@163.com
开始日期	2019-10-26	完成日期	2019-12-30

一、项目背景

以下是我根据作业要求，自己描述的情景，进行了一定的抽象：

企业 A 没有现金（或者存款），但是 A 的信誉很好，一家银行（或者金融机构）愿意给 A 担保，以帮助 A 解决资金周转困难的问题。于是银行给 A 开了一张凭证，A 可以将这张凭证当做支票一样使用。这样 A 就可以将凭证中的一部分“钱”转给另一家企业 B，用于购买企业 B 的货物，企业 B 既可以将“钱”用于向银行贷款，也可以继续传递给下一家企业 C。企业 B 之所以愿意相信这个凭证是有效的，是因为它有着企业 A 的信誉担保，银行也是认可的。同理，企业 C 也愿意相信这个凭证是有效的。大致情境图如下：



情景简化：

在上述情景中，可以进一步细化：

- 1、每一份合同需要一个 used 成员变量，它是被用来统计总额度被使用了多少。比如我们假设 A 给 B 的合同是 con_1，con_1 的总额度是 12000，初始的 used 肯定是 0。后来 B 给 C 了一个合同，假设是 con_2，con_2 的总额度是 5000，那么 con_1 的 used 就应该被更新为 5000。这样可以防止被使用的额度超过总额度的情况出现

$$\text{一份合同的余额} = \text{amount} - \text{used}$$

- 2、从上图中我们可以看出，有两种合同，信任合同和普通合同。其实他们的本质都是一样的，都有成员：开始时间、结束时间、两方、总额度。所以可以将其简化成一种。这个“两方”怎么简化呢？在信任合同中，可以理解成银行给了企业 A 20000 元；在普通合同中，可以理解成 A 给了 B 12000 元。这样二者就同样符合“from => to”模式。唯一不同的是，当合同到期，如果 from 方是银行，说明这是一份信任合同，需要在 to 方账户上扣钱；否则这是企业间的普通合同，需要在 to 方账户上加钱

这里解释一下，为什么最后的结算似乎跟 from 方无关。还是以上图为例，在所有合同到期后，A 应该失去 12000，因为他使用了信任合同中的 12000 给了 B，所以信任合同的 from 方（也就是银行）并没有起作用。B 应该得到 12000-5000=7000，也就是 A、B 之间的普通合同的余额。C 应该得到 5000，也就是 B、C 之间的普通合同的余额。所以普通合同的 from 方也没有起作用。正是 used 的使用导致了 from 的失效（不过不建议删除 from，from 可以用来查询交易历史）

- 3、贷款或者融资，可以简化为取款，这样就避免考虑贷款金额和利息等问题

二、 方案设计

(一) 存储设计

1、 结构体：

```
// 合同
struct Trust_contract
{
    uint start_time; // 合同开始时间
    uint end_time; // 合同结束时间
    address address_from; // 合同的 from 方
    address address_to; // 合同的 to 方
    uint amount; // 合同总额度
    uint used; // 被使用的额度
}
Trust_contract[] public contracts;
```

2、 变量：

变量类型	变量名	访问权限	存储内容
Address	Bank	Private	银行的地址
Mapping(address => uint)	Balance	Public	银行和企业的账户存款
Uint	Now_time	Private	现在的时间
Trust_contract[]	contracts	Public	合同

账户存款采用“地址 => 值”的映射关系，便于用户查询和函数调用

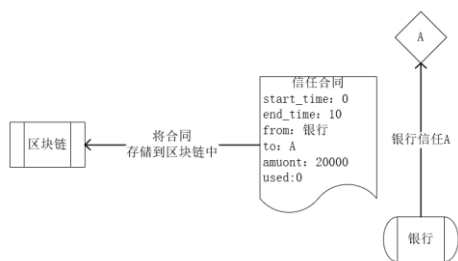
3、 函数：

函数就是围绕上述的结构体和变量进行设计。大概设计以下几个函数：

函数	需要完成的任务
构造函数	将变量 bank 设置成部署合约的用户，并设置一个初始账户存款 999999
查询账户存款	输入用户地址，可以查询用户的账户存款
存款	用户存钱，增加账户存款
签署信任合同	添加一个信任合同，from 方是银行
企业间交易	from 方可以利用已有的合同，发起交易。也就是添加一个普通合同
企业从银行“取款”	企业可以利用已有的合同，从银行“取款”（对应现实情况中的贷款或者融资）
企业还款+日期前进	前进到一个日期，完成并销毁所有到期的合同

(二) 数据流图

- 1、银行与企业 A 签订一份信任合同，总额度是 20000，相当于 A 有了一张总额度为 20000 的“空头支票”

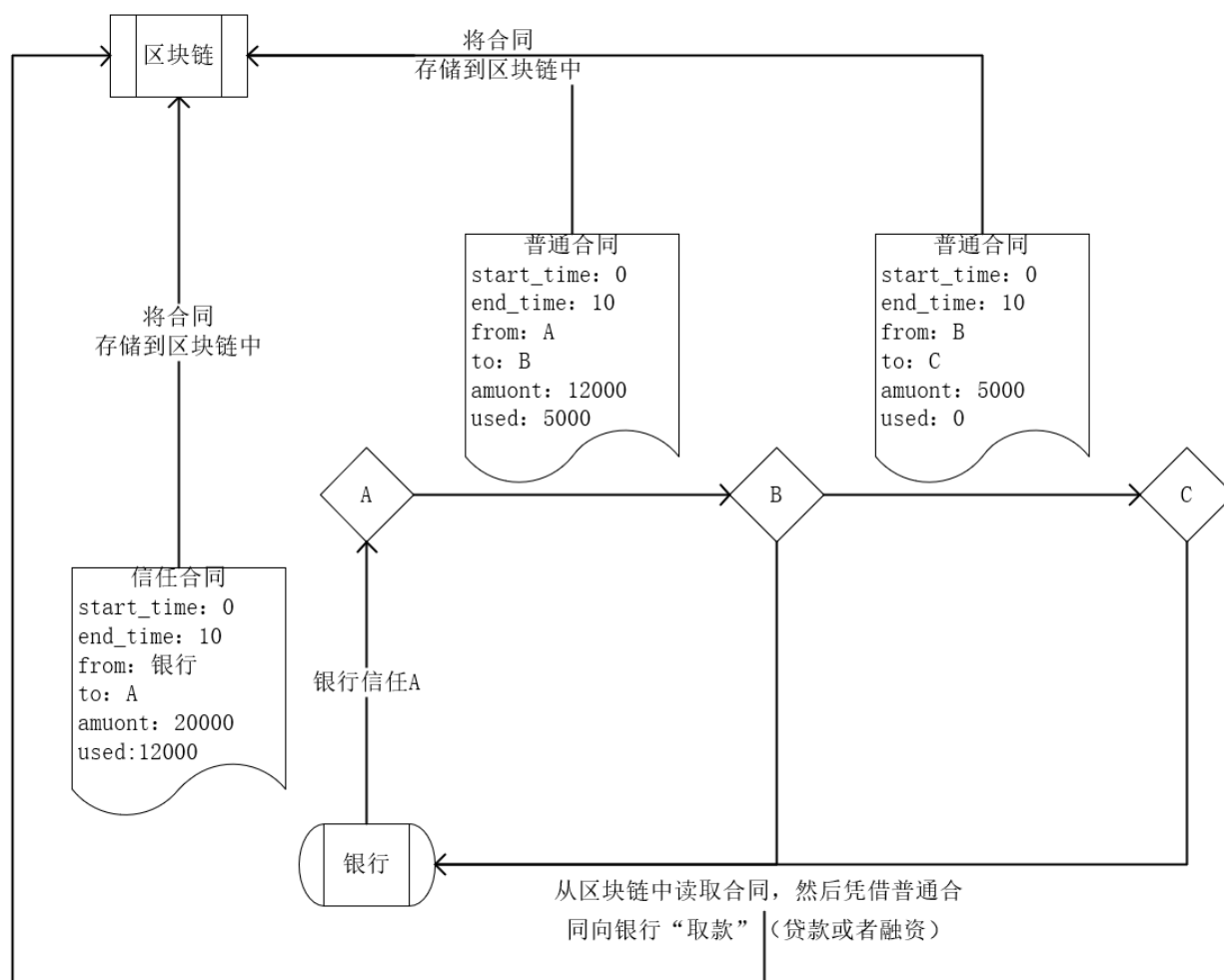


从数据结构的角度，新建一个 `Trust_contract` 对象：

```
{
  start_time = 0
  end_time = 10
  address_from = 银行的地址
  address_to = A 的地址
  amount = 20000
  used = 0
}
```

然后 push 到 contracts 中，也就是存储了这份合同

- 2、然后 A 凭借这份信任合同，可以向企业 B 购买货物，然后将信任合同中的部分额度转给 B。B 既可以将这部分额度用于向银行贷款或者融资（在我的设计中用“取款”简化），也可以继续转给下一家企业 C。如下图所示：



注意在使用额度的时候要更新相应合同的 `used`。在完成上图流程后，contracts 中就有了三个合同对象

主要是函数的逻辑实现，下面一个个介绍

将变量 bank 设置成部署合约的用户，并设置一个初始账户存款 999999:

2、查询账户存款函数

相较于上一次作业，我为了便于界面调用，新增了一个查询账户存款的函数：

3、存款

直接根据输入的用户地址，增加账户存款就可以了：

我在这里没有设置任何条件，也就是说银行也可以存钱

4、 签署信任合同

首先判断银行是否存在，以及时间是否合法（合同结束时间必须要比现在时间晚）：

然后添加信任合同。一份合同的参数我在之前已经提过了，其中合同开始的时间就是现在的时间，合同的 from 方是银行，合同的 to 方是交易的发起方（msg.sender()），被使用的额度是 0。所以需要用户输入结束的时间和总额度：

- 5 -

```

// 添加信任合同
Trust_contract memory t=Trust_contract(now_time,end_time,bank,msg.sender,money,0);
contracts.push(t);
}

```

5、企业间交易

一次企业间的交易，就建立一份普通合同。合同的 from 方就是交易发起者，to 方则需要用户指定。普通合同的总额度也要用户指定

考虑这么一个情形，企业 A 得到了银行的一份信任合同，然后企业 A 与企业 B 交易，需要一份普通合同。这是很容易实现的，计算出信任合同的余额，然后与普通合同的总额度比较，如果余额足够，直接使用相应的额度；如果余额不够，拒绝即可

但是在现实生活中，合同肯定是多种多样的，情况也是非常复杂。我考虑了以下几种情况：

- (1) 企业 A 可能从多家银行获得了信用合同，可能其中一份信用合同的余额是不够的，但是多份合同的余额加在一起就足够了。所以我们需要计算企业 A 的所有余额的总和
- (2) 假设企业 A 与企业 B 达成交易，那么企业 B 就只有普通合同，没有信任合同，但是两种合同的效益应该是一样的。所以计算某家企业所有余额的总和，不应该只考虑信任合同，还应该考虑普通合同。这就体现出之前将两种合同合并为一种结构体的好处了，合同的 to 方才有权利使用这份合同的余额
- (3) 不同合同的开始时间和结束时间都不一样，默认继承时间。举个例子：假设现在的时间是 8，企业 A 使用一份时间范围在 0-10 的合同和一份时间范围在 5-15 的合同，向公司 B 发起交易（假设余额足够），那么就新建两份普通合同，其中一份时间范围是 0-10，另一份是 5-15

考虑清楚以上情况，就很容易写出代码了：

```

// 企业间交易的函数
function deal(address address_to,uint money)public
{
    // 判断交易的 to 方是不是 bank，如果是则终止交易
    if(address_to==bank)return;

    // 遍历发起交易的企业的合同，判断余额是否足够
    uint mount=0; // mount 用于储存总余额
    bool flag=false;
    for(uint i=0;i<contracts.length;i++)
    {
        if(contracts[i].address_to==msg.sender)mount+=(contracts[i].amount-contracts[i].used);
        if(mount>money)

```

```

    {
        flag=true;
        break;
    }
}

// 如果总余额不足，终止交易
if(flag==false)return;

// 开始交易，遍历所有合同。合同的 to 方才有权利使用这份余额
// money 是用户输入的值，代表两家企业间交易的额度
for(i=0;i<contracts.length;i++)
{
    if(contracts[i].address_to==msg.sender)
    {
        // 如果一份合同的余额就足够，直接给这份合同的 used 加上 money，并新建一份额度
        // 为 money 的普通合同，结束遍历
        if(contracts[i].amount-contracts[i].used>=money)
        {
            contracts[i].used+=money;

            Trust_contract memory t=Trust_contract(now_time,contracts[i].end_
time,msg.sender,address_to,money,0);
            contracts.push(t);

            break;
        }
        // 如果一份合同的余额不够，就用光这份合同的余额，并新建一份额度为所用余额的
        // 普通合同，更新 money 用于下一个循环
        else
        {
            money--(contracts[i].amount-contracts[i].used);

            t=Trust_contract(now_time,contracts[i].end_time,msg.sender,addres
s_to,contracts[i].amount-contracts[i].used,0);
            contracts.push(t);

            contracts[i].used=contracts[i].amount;
        }
    }
}
}
}

```

6、企业从银行“取款”（“取款”相当于贷款或者融资）

逻辑上跟企业间交易是一样的，只不过企业间交易是体现在新建一份普通合同上，“取款”体现在企业的账户存款增加了。代码和上一个函数很相似，只需要把新建一份普通合同的部分，修改成增加账户存款就可以了，这里就不展示了

7、企业还款 + 日期前进

前进到某个日期，某些合同就过期了。对于这些过期的合同，如果是信任合同，也就是合同的 from 方是银行，就应该由合同的 to 方还款，还钱的数额是这份信任合同的 used。注意不要搞成了 amount，因为 amount 并没有全部被用掉，真正被用掉的部分是 used，只需要还款这部分就可以了。对于普通合同，则由合同的 to 方获取合同的余额，也就是 $\text{amount} - \text{used}$

所以函数的过程应该是：计算总欠款并与相应账户存款比较，如果存款不够就终止；如果存款足够，就给信任合同的 to 方扣钱，给普通合同的 to 方加钱，然后删除合同、更新时间即可。代码如下：

```
// mount 用于储存总欠款
mapping(address => uint) mount;
// 企业还款的函数 + 日期前进的函数
function repay(uint time) public
{
    // 判断前进的日期是否大于现在的日期
    if(time <= now_time) return;

    // 初始化 mount
    for(uint i=0; i<contracts.length; i++) mount[contracts[i].address_to]=0;
    // 计算总欠款
    for(i=0; i<contracts.length; i++)
    {
        if(contracts[i].end_time < time)
        {
            if(contracts[i].address_from == bank)
            {
                mount[contracts[i].address_to] += contracts[i].used;
            }
        }
    }
    // 比较总欠款和账户存款，如果存款不够就不能还款 + 日期前进
    for(i=0; i<contracts.length; i++)
    {
        if(mount[contracts[i].address_to] > balance[contracts[i].address_to])
        {
            return;
        }
    }

    for(i=0; i<contracts.length; i++)
    {
        if(contracts[i].end_time < time)
        {
            // 如果合同的 from 方是 bank，这是一份信任合同，就从 to 方的账户上扣钱
```



```

        if(contracts[i].address_from==bank)
        {
            balance[contracts[i].address_to]-=contracts[i].used;
            // 不要忘记删除合同
            delete contracts[i];
        }
        // 否则这是一份普通合同，就在 to 方的账户上加钱
        else
        {
            balance[contracts[i].address_to]+=(contracts[i].amount-
contracts[i].used);
            // 不要忘记删除合同
            delete contracts[i];
        }
    }
}

now_time=time;
}

```

对照一下大作业要求实现的四个功能：

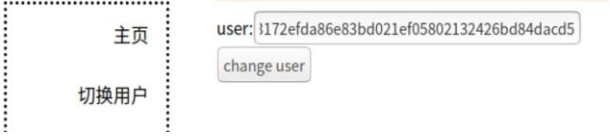

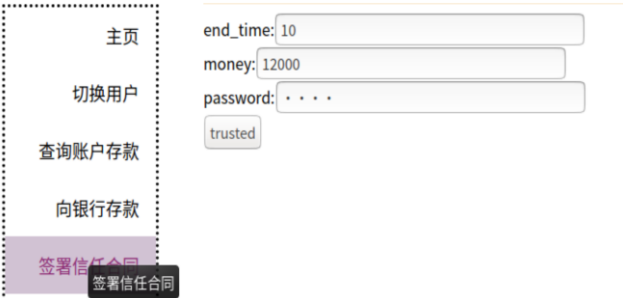
要求实现的功能	我实现的方法
功能一：实现采购商品—签发应收账款交易上链。例如车企从轮胎公司购买一批轮胎并签订应收账款单据	在企业间交易函数中，可以根据信用合同新建一个普通合同
功能二：实现应收账款的转让上链，轮胎公司从轮毂公司购买一笔轮毂，便将于车企的应收账款单据部分转让给轮毂公司。轮毂公司可以利用这个新的单据去融资或者要求车企到期时归还钱款	1、在企业间交易函数中，可以根据普通合同新建一个普通合同 2、在“取款”函数中，可以根据合同向银行进行“取款”（相当于融资或者融资） 3、在还款 + 日期前进函数中，可以在合同到期时，要求车企还款
功能三：利用应收账款向银行融资上链，供应链上所有可以利用应收账款单据向银行申请融资	在“取款”函数中，可以根据合同向银行进行“取款”（相当于融资或者融资）
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	在还款 + 日期前进函数中，可以在合同到期时，所有下游企业收款

完整代码请看根目录下的 Bank.sol

三、 功能测试

如果需要看 **WeBASE** 的测试效果，可以看我上一次作业的报告。这一次作业的功能测试是在我自己写的框架下跑的，框架的设计在下一部分“界面展示”。功能测试的截图及分析如下：

<p>1、界面展示</p> <p>界面的左侧是功能栏，分别实现了如图所示的几种功能。然后点击每一种功能，都会在右侧显示输入界面，根据要求输入参数，点击按钮获取结果</p>	<div><h3>切换用户</h3><div><div>主页</div><div>切换用户</div><div>查询账户存款</div><div>向银行存款</div><div>签署信任合同</div><div>企业间交易</div><div>企业从银行取款</div><div>还款</div></div><div>user: 1172efda86e83bd021ef05802132426bd84dacd5 change user</div></div>
<p>2、输入合约地址</p> <p>点击左侧的“主页”，输入合约地址，就可以调用合约了</p>	<div><h3>合约地址</h3><div><div>主页</div><div>切换用户</div></div><div>contract address: d33338b95398c79adb13d462e4c01bcb81b2721 confirm</div></div>
<p>3、切换用户为银行</p> <p>点击左侧的“切换用户”，输入用户地址，就可以切换成该用户。如图所示，输入银行的地址，切换成银行</p>	<div><h3>切换用户</h3><div><div>主页</div><div>切换用户</div></div><div>user: if8447a787c90dd0022015c7367c5bd68a711458 change user</div></div>
<p>4、查询账户存款</p> <p>任何用户都可以查询其他用户的账户存款。点击左侧的“查询账户存款”，输入账户地址，就可以得到</p>	<div><h3>查询账户存款</h3><div><div>主页</div><div>切换用户</div></div><div>address: if8447a787c90dd0022015c7367c5bd68a711458 check balance</div></div>

<p>账户存款。首先测试银行的存款，得到结果是 999999（这是合约的构造函数规定的）</p>	<p>balance: 999999</p>
<p>5、查询账户存款 继续查询其他企业的账户存款，以企业 A 为例</p>	<p>balance: 0 所有企业的账户存款都是 0</p>
<p>6、存款 切换用户为企业 A，并给企业 A 存款 20000</p>	<p>(1) 切换用户为企业 A 切换用户</p>  <p>(2) 存款 20000 存款</p> 
<p>7、信任合同 银行和企业 A 签署亲人一份结束时间是 10、总额度为 12000 的信任合同。在签署信任合同的时候，要求用户输入密码（默认密码是 1234），这个密码有点像是银行发给企业的验证码</p>	<p>签署信任合同</p> 

<p>8、普通合同</p> <p>企业 A 在签署信任合同后，他可以凭借这份信任合同和其他企业签署普通合同，比如和企业 B 签署一份总额度为 5000 的普通合同，相当于 A 花费了 5000 从企业 B 这里购买货物。输入企业 B 的地址和合同额度</p>	<div> <div> 主页 切换用户 查询账户存款 向银行存款 签署信任合同 企业间交易 </div> <div> <h3>企业间交易</h3> <div> address_to: 0xb34022efd812f8eddb7f1f1727c2ff474888e7f0 </div> <div> money: 5000 </div> <div>deal</div> </div> </div>
<p>9、企业从银行“取款”</p> <p>这个取款就相当于企业 B 凭借着和企业 A 签署的合同，向银行贷款或者融资。我的合约会自动帮用户从他的合同中使用相应的额度</p>	<div> <div> <p>(1) 切换用户为企业 B</p> <h3>切换用户</h3> <div> <div> 主页 切换用户 </div> <div> user: 0xb34022efd812f8eddb7f1f1727c2ff474888e7f0 <div>change user</div> </div> </div> </div> <p>(2) 企业 B 从银行取款 2000</p> <div> <h3>企业从银行取款</h3> <div> <div> money: 2000 </div> <div>loan</div> </div> </div> <p>(3) “取款”之后，查询企业 B 的账户存款，发现已经由 0 变成了 2000</p> <div> balance: 2000 </div> </div>
<p>10、还款</p> <p>目前所有的合同都是在时间为 10 的时候到期，所以我们将时间前进到 20，检查一下是否所有合同都被</p>	<p>(1) 时间前进到 20</p>

按时完成了

还款

time: 20

repay

主页

切换用户

查询账户存款

向银行存款

签署信任合同

企业间交易

企业从银行取款

还款

(2) 在合约设计的部分我解释过，如果账户存款不够用来还款的话，时间不会前进。目前企业 A 的存款是 20000，而信用合同总额度 12000，被使用的部分只有 5000，所以还款完全不是问题。企业 A 的存款应该减少 5000，企业 B 的存款应该增加 5000

```
000000000000000000000000  
0000","transaction  
edRaw":"0x5add"}  
  
balance: 15000  
[ ]  
  
000000000000000000000000  
0000","transaction  
edRaw":"0x5add"}  
  
balance: 5000
```

查询账户存款

address: 1172efda86e83bd021ef05802132426bd84dacd5

check balance

查询账户存款

address: 0xb34022efd812f8eddb7f1f1727c2ff474888e7f0

check balance

由上图可以得知，企业 A 的账户余额是 $20000 - 5000 = 15000$ ，企业 B 的账户余额是 5000

综上所述，测试成功。完整的过程请看演示视频

四、 界面展示



界面的左侧是功能栏，分别实现了如图所示的几种功能。然后点击每一种功能，都会在右侧显示输入界面，根据要求输入参数，点击按钮获取结果

这次的界面的设计使用了 express 框架。这是一个基于 node.js 的 web 应用开发框架。

这次作业的源代码在 Bank 文件夹中。它的目录结构是：

- bin
- node_modules
- public
- routes
- views
- app.js
- package.json
- package-lock.json

1、Routes

routes 文件夹中的 index.js 文件是路由文件。它生成一个路由示例来捕获页面的 GET 请求和 POST 请求，以帮助我们获取界面和返回参数。以 index.js 中，关于查询账户存款的部分为例：

```
// GET 请求
router.get('/balance',
function(req, res) {
    res.render('balance', {
        title: '查询账户存款'
    });
});

// POST 请求
router.post('/balance',
function(req, res) {
    // 一个变量用于存储需要使用的合约函数参数
    var account_checked = req.body.account_checked,

    http = require('http');
    // 将合约函数参数以数组的形式存储下来（因为参数可能不止一个，类型也不相同）
    var arr = [];
    arr.push(account_checked);

    // 为了使用 WeBASE 的 API，需要将 API 的参数以数组的形式存储下来
    var post_data = {
        "useAes": false,
        "user": user,
        "contractName": "Bank",
        "contractAddress": contract_address,
        "funcName": "check_balance",
        "funcParam": arr,
        "groupId": "1"
    };

    // 然后将 API 的参数数组转成 json 格式
    var content = JSON.stringify(post_data);

    var options = { hostname: '127.0.0.1',
        port: 5002,
        path: '/WeBASE-Front/trans/handle',
        method: 'POST',
        headers: {
            "Content-type": "application/json"
        }
    };

    var req = http.request(options,
    // 将得到的 balance 输出到控制台上
    function(res) {
        var _data = '';
```

```

    res.on('data',
    function(chunk) {
        _data += chunk;
    });












    res.on('end',
    function() {
        console.log("result:", _data)
        _json = JSON.parse(_data)
        console.log("\nbalance: ", parseInt(_json.output))
    });
});

// 发送 API 的参数
req.write(content);
req.end();
// 更新界面
res.redirect('/balance');
});

```

2、Views

index.js 路由文件所获取的界面正是被储存在 views 文件夹中：

 balance.ejs	
 deal.ejs	
 deposit.ejs	
 error.ejs	index.ejs 是主页
 footer.ejs	header.ejs 和 footer.ejs 是每个界面所共有的部分，其中 header.ejs 就是左侧的功能栏
 header.ejs	
 index.ejs	其他的文件都对应着一个功能，比如 balance.ejs 文件是用来查询账户存款的
 loan.ejs	
 repay.ejs	
 trusted.ejs	
 user.ejs	

以 balance.ejs 为例，简单说明一下功能的实现：

```

<%- include header %>
<form method="post">
    address:<input type="text" name="account_checked"/><br />
    <input type="submit" value="check balance"/>
</form>

```



```
<%- include footer %>
```

第一行和最后一行就是所有界面共用的部分，也就是左侧的功能栏

中间的部分用来捕获 POST 请求，包括一个 `account_checked` 变量 和 一个按钮。点下按钮，`account_checked` 变量就被返回

五、 心得体会

通过这次大作业，我深入了解了区块链的理论和 `solidity` 的各种语法，还初步掌握了 `express` 框架的使用。虽然这次大作业花费了我非常多的时间，我也遇到了很多困难。但是在完成的那一刻，我是非常高兴且充实的。而且通过这次大作业，我也学会了如何搭建一个区块链应用