

區塊鏈技術服務產學聯盟

食品安全信任平台

以太坊實作食品履歷以及原物料管控系統

開發文件報告

指導教授：黃明祥

目錄

軟體介紹.....	3
• 動機與需求.....	3
• 功能概述.....	3
• 注意事項.....	3
• 版本環境參考.....	3
前端環境.....	4
後端環境部屬.....	6
• Python server.....	6
• Block Chain (Geth)	11
• 智慧合約說明.....	14
執行流程.....	22
• 首頁.....	22
• 店家端功能.....	22
• 顧客端功能.....	27

軟體介紹

• 動機與需求

現代社會發展趨勢，大多民眾以外食為主，食品安全也成為一件重要的事。本軟體之目的在於建構一個 app，提供店家與顧客雙向的服務，搭建一個橋樑，以顧客的角度來說：能透過查看我們 app，檢視合作商家每天的資料更新，減少對於食品、餐點的安全疑慮。以店家的角度而言：透過系統化的資料上傳，由我們的技術為其統整，一目了然的資料呈現進一步達到降低管理食材原料的成本，可以有效的管控食材進貨量並估算未來銷售策略。

以半透明的食材管理過程讓消費者能夠選擇更為健康有保障的店家，甚至能以此推動店家間互相砥礪。並且在第二版我們還推出了以 ERC20 技術為底的虛擬亞大幣，做為該軟體的回饋機制，消費者紀錄飲食就能獲得虛擬貨幣，可在合作店家消費折抵，店家再將收集到的虛擬貨幣轉換為現金。

• 功能概述

顧客方：查看店家的開關店的狀態、食材狀態、清潔狀態。

店家方：在食材管理部分，透過掃描食材上的 QR CODE，由後台去依據掃描次數判定其狀態(進貨、啟用、用盡)的時間，更好地管理並呈現食材的乾淨與新鮮度。

• 注意事項

- Geth 的版本不同會導致連線的參數不一樣。
- Solidity 每一版也會有更改/刪除的參數。
- 挖礦一直出現”Generating DAG in progress epoch=0 percentage=1 elapsed=3.448s”的錯誤，可以等到 DAG percentage=100，就會開始，沒成功的話可能跟創世區塊設定有關。
- Cordova run 無法執行，除了安裝問題也有可能是電腦的執行原則設定 (server 為 RemoteSigned)。

• 版本環境參考

- Windows 10
- Node : 13.12.0、npm : 6.14.4
- Gradle : 7.11
- Bootstrap : 5.0.2
- Solidity : 0.5.12
- Geth : 1.10.8
- Golang : 1.17
- Python : 3.7

前端環境

APP version : 1.2.1

使用 cordova 編譯，使用到的套件如下：

```
"license": "Apache-2.0",
"devDependencies": {
  "cordova-android": "^9.1.0",
  "cordova-browser": "^6.0.0",
  "cordova-plugin-add-swift-support": "^2.0.2",
  "cordova-plugin-barcodescanner": "^0.7.4",
  "cordova-plugin-compat": "^1.2.0",
  "cordova-plugin-whitelist": "^1.3.5",
  "cordova-sqlite-storage": "^6.0.0"
},
```

1.0.0 版本的時後以手刻為主，1.1.0 時開始套入一部分 bootstrap 做應用。

Cordova 環境安裝

- 安裝 [node.js](#)：選擇安裝版本 並下載.msi 檔
- 安裝 Cordova：`npm install -g cordova`
- 創建一個新的專案：`cordova create [Project name]`

```
D:\
λ cordova create food_BlockChain
? May Cordova anonymously report usage statistics to improve the tool over time? Yes

Thanks for opting into telemetry to help us improve cordova.
Creating a new cordova project.
```

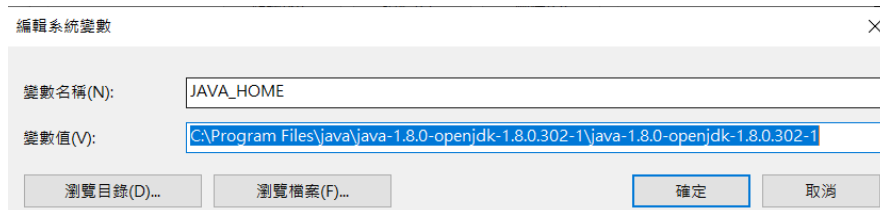
- 再來進到剛剛創建的專案資料夾中，依照需求新增平台（android、browser）
- `cordova platform add android`

```
D:\food_BlockChain_Cordova (io.cordova.hellocordova@1.0.0)
λ ls
config.xml  node_modules/  package.json  package-lock.json  platforms/  plugins/  www/

D:\food_BlockChain_Cordova (io.cordova.hellocordova@1.0.0)
λ cd platforms\

D:\food_BlockChain_Cordova\platforms
λ ls
android/
```

- 執行 Android app 需要安裝 [Gradle](#)(v7.11)、[android studio](#)
- Gradle 下載完解壓縮，檔案一般在下載裡，到 C:\Program Files 裡建立該 sdk 檔名之資料夾，把檔案拖曳進去
- 複製到\bin 的路徑
- 檔案總管>本機右鍵>內容>相關設定>進階系統設定>環境變數（N）>系統變數>Path>新增一個新的貼上
- 回到系統變數>點選新增>輸入 JAVA_HOME>貼上路徑並把\bin 拿掉



- 下載 Java SE 8，記得設定環境變數（msi 檔則不用）
- 輸入 `cordova requirements` 可以檢查

```
D:\projects\WebApp\food_BlockChain_Cordova (io.cordova.hellocordova@1.0.0)
λ cordova requirements

Requirements check results for android:
Java JDK: installed 1.8.0
Android SDK: installed true
Android target: installed android-31,android-29
Gradle: installed C:\Program Files\gradle\gradle-7.1.1-all\gradle-7.1.1\bin\gradle.BAT

Requirements check results for browser:
```

連線到 python 端的功能

- 以 WebSocket 通道連線，在紅框內填入 python 端 IP 與開啟的 port 號
- 將資料（{"Main: [Contract Name]", "Type: [Function Name]"}）包裝成 JSON 格式

```
function onload(){
  ws = new WebSocket("ws://192.168.0.105:6012");
  ws.onopen = function () {
    console.log('open');
    sendData["Main"] = "storeContract";
    sendData["Type"] = "firstLogin";
    let jsonData = JSON.stringify(sendData);
    ws.send(jsonData);
  };
};
```

- 建立連線並傳送指定 function 要求後會再傳送資料
- WebSocket 的 `ws.onmessage` 函式會取得後端回傳的資料，可以是簡單字串或是陣列等等。

```
var check = event.data;
if (check=="true"){
  localStorage.address = address;
  localStorage.pwd = password;
  alert('登入成功');
  window.location.href='loginafter.html';
}

ws.send(address);
ws.send(account);
ws.send(password);
```

後端環境部屬

- Python server

Python 套件：

```
pip install web3
```

```
pip install websockets
```

註 1：web3 的下載碰到 “error: Microsoft Visual C++ 14.0 is required.”，是參照這個[網頁](#)，下載了對應了 Visual Studio 跟相關套件。

第一次部屬和區塊鏈連線時出現過這個錯誤，後來發現是創世區塊有少進行參數的設定，故而整個鏈又重新設定。（創世區塊重新設定即代表區塊鏈一定要重新初始化）

```
ValueError: {'code': -32000, 'message': 'invalid opcode: SHR'}  
PS D:\Blockchain>
```

新增的參數："byzantiumBlock": 0 與 "constantinopleBlock": 0

Python 端要和區塊鏈網路連結進行合約交易，也要連線前端以 cordova 部屬的 app，在初步開發時將兩端的連線寫在同一份檔案，後來由於功能擴增有進行改變。

1.1.0 時將 py 功能分為 "Connection.py" 和 "Contract.py"，前者用來進行和前端的連線，並將要求傳送到第二個 py 檔案，進行對應的區塊鏈交易。

註 2：前端只會和 "Connection.py" 溝通，再由 "Connection.py" 呼叫 "Contract.py" 的功能

更改過程第一步改動了原本的呼叫方式。

- 在 "Contract.py" 裡以 class 分割兩份合約→storeALL、clientALL
- 更改了 "Connection.py" 裡的呼叫模式。由各頁 js 建立 websocket 連線時，同步傳送此時應使用哪份合約之要求（簡單字串）
- js 傳來的資料分別為 "Main: [Contract Name]"、"Type: [Function Name]"
- 取得後為 string，it's not callable → 預先使用 dict 存放 method name

```
allName = {  
    'storeContract': cf.storeContract,  
    'clientContract': cf.clientContract  
}
```

- cf 為 import Contract 在此頁的代稱

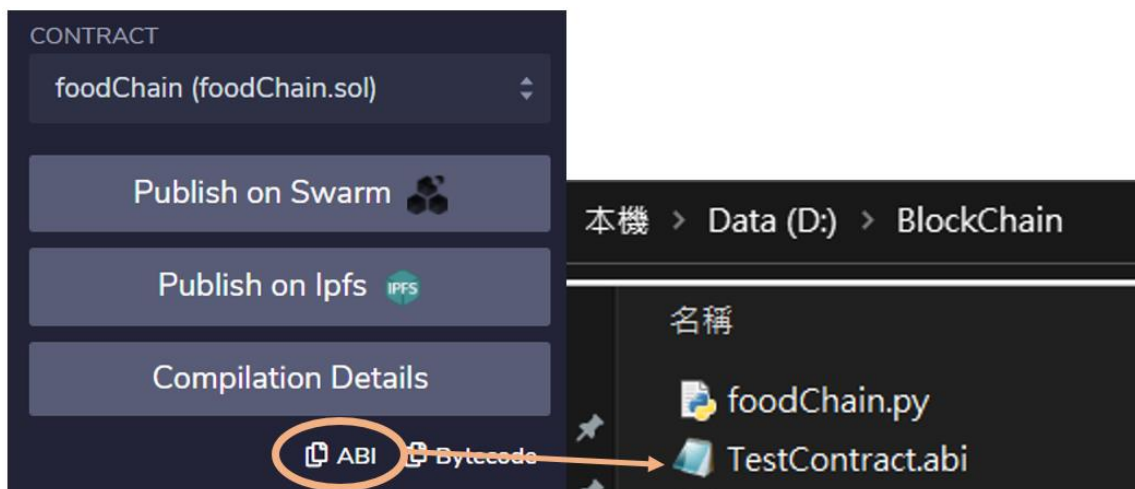
第二步則是更改了 js 連線 py 的資料。

- {"Main: [Contract Name]", "Type: [Function Name]}"}
- 需要經過單位轉換，從 str → json 經由 websocket 傳送，到 py 再解構

```
str = JSON.loads(str)
```

連接區塊鏈網路 (Contract.py)

- 定義 web3，開發過程中由於 python 和區塊鏈節點位於同一台 server，所以在程式中使用本地端 127.0.0.1，port 號則與 geth 設定的相同。
- 在 remix 上複製合約的 abi 檔另外儲存，在程式中透過開檔的方式引入。Abi 的版本必須和合約同步，裡面有的函數才能在引入後被使用。



- 設定合約位址，使用 remix 上 deploy 後的合約位址進行連線。
- 設定發布該合約與執行的帳戶。

```
1 from web3 import HTTPProvider, Web3
2 import json as JSON
3 from datetime import datetime
4 import os
5 import numpy as np
6
7 Zoeyunnnn, 2 months ago | 3 authors (zzz20002026 and others)
8 class storeContract:
9     def __init__(self):
10         print("storeContract Success")
11         self.w3 = Web3(HTTPProvider('http://127.0.0.1:8080'))
12         self.count = 0
13         with open("storeALL.abi") as f:
14             self.temp_abi = JSON.load(f)
15
16         # 設定合約位址
17         self.contract_addr = self.w3.toChecksumAddress('0xE5e21344Df3791ebF701c75c624edb59302A230B')
18         self.contract = self.w3.eth.contract(address=self.contract_addr, abi=self.temp_abi)
19         # 設定帳號位址
20         self.account = self.w3.toChecksumAddress("0xb93E7ba12f4D6D9AAF0974a676f992ac5EE15969")
```

再來以食材進貨的功能舉例說明。

- 下圖為合約的函式，要設定進貨時間需要 id 跟 address 兩個參數。

```
//食材進貨輸入
function setDeliverTime(string memory _id, address _storeAddress) public onlyStore(_storeAddress){
    require(stores[_storeAddress].foodLists[_id].isVaild != true);
    stores[_storeAddress].foodLists[_id] = foodList({
        id: _id,
        storeAddress: _storeAddress,
        status: 1,
        inputTime: block.timestamp,
        clearTime: 0,
        isVaild: true
    });
    stores[_storeAddress].i = stores[_storeAddress].i+1;

    //將時間與食材資料填入 timeLists
    timeLists[_storeAddress].foodID.push(_id);
    timeLists[_storeAddress].inputTime.push(stores[_storeAddress].foodLists[_id].inputTime);
}
}
```

- 建立交易 (build Transaction) 的部分
 - ◆ 首先要先透過 web3 函式庫裡的 `toChecksumAddress()` 來將前端 javascript 傳過來的 address 轉換成真正的以太坊地址。
 - ◆ 定義 id 跟 address (紅線部分)
 - ◆ chainID 是 geth 創世區塊檔案(genesis.json)裡面的設定，要跟它一樣不然會連接不到！
- 簽名並傳送交易的部分
 - ◆ geth 節點裡面會有 keystore 這個資料夾（存放私鑰的加密檔案）
 - ◆ 目前我們是手動改檔名，將檔名改為 address，然後將要使用這個函式的使用者 address 從前端傳送過來 python，去找 keystore 裡面對應的檔案，解密後就可以簽屬並發送這筆交易。

```
# 設定食材進貨時間
def setDeliverTime(self, id, address, password):
    address = self.w3.toChecksumAddress(address)
    estimate_gas = self.contract.functions.setDeliverTime(id, address).estimateGas()
    nonce = self.w3.eth.getTransactionCount(address)
    txn = self.contract.functions.setDeliverTime(id, address).buildTransaction({
        'chainId': 428,
        'gas': estimate_gas,
        'gasPrice': self.w3.toWei('1', 'gwei'),
        'nonce': nonce
    })

    #設定私鑰
    path = "D:/Blockchain/node1/keystore/"
    x = os.path.join(path, address)
    storeKey = password
    with open(x) as keyfile:
        encrypted_key = keyfile.read()
        private_key = self.w3.eth.account.decrypt(encrypted_key, storeKey)
        print(bytes.hex(private_key))
        key = bytes.hex(private_key)
        signed_txn = self.w3.eth.account.signTransaction(txn, key)

    tx_hash = self.w3.eth.sendRawTransaction(signed_txn.rawTransaction)
    # print('0x'+bytes.hex(tx_hash))
    return "Success"
```


以 WebSocket 連線前端 (Connection.py)

- WebSocket server 的主要結構
- 參照下圖，7~11 行跟 16~21 行是在判斷要使用 Contract.py 的哪個合約
- 中間透過資料內容去判斷是哪個 js 檔傳過來的，要做什麼事?等等

```
1 import Contract as cf
2 import json as JSON
3 import asyncio
4 import websockets
5 import time
6
7 allName = {
8     'storeContract': cf.storeContract,
9     'clientContract': cf.clientContract,
10    'asiaToken': cf.asiaToken
11 }
12
13 async def echo(websocket, path):
14     connected = set()
15     connected.add(websocket)
16     try:
17         str = await websocket.recv()
18         str = JSON.loads(str)
19
20         contractName = str["Main"]
21         func = allName[contractName]
22         contract = func()
23
24         # 這邊開始寫if else來判斷是哪個頁面，要做什麼事
25
26     finally:
27         connected.remove(websocket)
28
29 async def main():
30     async with websockets.serve(echo, "192.168.0.123", 6012):
31         await asyncio.Future() # run forever
32
33 if __name__ == "__main__":
34     asyncio.run(main())
```

1.2.0 新增了亞大幣 (AsiaToken) 的功能。

透過官方的 ERC20 標準而發布，該範例內含許多不同的功能函式，所以在合約上

另外獨立，發布時選擇



。

定義 transfer 函式用來發布測試幣。每次使用主帳號（預設官方帳戶）發送\$10 token。

```

#發送測試幣
def transfer(self, address):
    # print(self.contract.functions.totalSupply().call())
    address = self.w3.toChecksumAddress(address)
    # estimate_gas = self.contract.functions.transfer(address, 10).estimateGas({'from': address})
    nonce = self.w3.eth.getTransactionCount(self.account)
    txn = self.contract.functions.transfer(address, 10).buildTransaction({
        'chainId': 428,
        'gas': 5000000,
        'gasPrice': self.w3.toWei('1', 'gwei'),
        'nonce': nonce
    })
    # print(txn)
    #設定私鑰
    with open(r'D:\Blockchain\node1\keystore\0xb93e7ba12f4d6d9aaf0974a676f992ac5ee15969') as keyfile:
        encrypted_key = keyfile.read()
        private_key = self.w3.eth.account.decrypt(encrypted_key, '1234wxyz')
        # print(bytes.hex(private_key))
        key = bytes.hex(private_key)
        signed_txn = self.w3.eth.account.signTransaction(txn, key)

    tx_hash = self.w3.eth.sendRawTransaction(signed_txn.rawTransaction)
    print('0x'+bytes.hex(tx_hash))
    return "Success"

```

在前端 Custom-info.js 裡有取得測試幣的按鈕，當按下按鈕會觸發 js 裡的 `getCoin()` 這個函式，送出 jsonData 跟顧客的 address。

```

function getCoin(){
    var ws = new WebSocket("ws://192.168.0.123:6012");
    ws.onopen = function () {
        console.log('open');
        sendData["Main"] = "asiaToken";
        sendData["Type"] = "transfer";
        let jsonData = JSON.stringify(sendData);
        ws.send(jsonData);

        // var ad = "0xDf11D1f32DAF325aa4Ce385A08c33F4D05Ab5FB9";
        // ws.send(ad);
        ws.send(localStorage.address);
    };

    ws.onmessage = function (event) {
        console.log(event.data)
        setTimeout(getbalance, 10000);
    };
}

```

Python 端接收到 jsonData 的資料後，要先 `approve()`，允許使用者(`check[0]`)轉多少錢給(`check[2]`)，再來會將從 js 得到的 address 丟到 Contract.py 的

`transferFrom()`裡，當交易成功後、資料上鏈，就會回傳 Transfer success 給 js，然後在前端頁面就能在亞太幣那欄看到自己的餘額。

```
elif str["Type"] == "transferFrom":
    check = []
    async for message in websocket:
        n = f"{message}"
        # print(n)
        check.append(n)
        print(check)
        if len(check)==4:
            state = contract.approve(check[0], int(check[2]), check[3])
            if(state=="Success"):
                result = contract.transferFrom(check[0], check[1], int(check[2]), check[3])
                await websocket.send(JSON.dumps(result))
            check.clear()
```

```
# customer-info.js - 顧客資訊
elif str["Type"] == "transfer":
    # await websocket.send("check")
    check = []
    async for message in websocket:
        n = f"{message}"
        print(n)
        check.append(n)
        if len(check)==1:
            address = check[0]
            # coin = check[1]
            contract.transfer(address)
            await websocket.send(JSON.dumps("Transfer success"))
            check.clear()
```



1.2.1 亞太幣測試的時候，發現在 Contract.py 裡合約部分關於參數 nonce 的設定一直有 bug，之前都設定 self.account 導致不同使用者發送交易卻一直使用同個主 address，應該改成 address 後就會變成不同的使用者發送交易。

```
address = self.w3.toChecksumAddress(add_from)
# print(type(address))
estimate_gas = self.contract.functions.approve(address, coin).estimateGas()
# print(estimate_gas)
nonce = self.w3.eth.getTransactionCount(address)
# print(nonce)
```

• Block Chain (Geth)

- 下載 [Golang](#)，開發版本為 1.17。安裝後透過 `go version` 確定成功。

- [官網](#)下載 Geth，開發版本為 1.10.8。安裝時 geth 和 tools 都要勾選。
- 安裝後如下圖，為開發方便將其全部複製到新資料夾（D:BlockChain/）中

 abigen.exe	2021/8/12 下午 03:22	應用程式	36,087 KB
 bootnode.exe	2021/8/12 下午 03:22	應用程式	36,792 KB
 clef.exe	2021/8/12 下午 03:22	應用程式	50,890 KB
 evm.exe	2021/8/12 下午 03:23	應用程式	36,769 KB
 geth.exe	2021/8/12 下午 03:23	應用程式	61,378 KB
 puppeth.exe	2021/8/12 下午 03:23	應用程式	23,421 KB
 rlpdump.exe	2021/8/12 下午 03:23	應用程式	2,439 KB
 uninstall.exe	2021/8/21 下午 02:54	應用程式	123 KB

- 新增一個創世區塊 genesis.json，開發設置如下，各項參數規則與意涵可參考 [官網](#)。

```
{
  "config": {
    "chainID": 428,
    "homesteadBlock": 0,
    "eip150Block": 0,
    "eip150Hash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "eip155Block": 0,
    "eip158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0
  },
  "nonce": "0x0000000000000042",
  "difficulty": "0x400",
  "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp": "0x00",
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "extraData": "",
  "gasLimit": "0x2fefd8",
  "alloc": {
    "0x0000000000000000000000000000000000000000000000000000000000000001": {
      "balance": "123"
    },
    "0x0000000000000000000000000000000000000000000000000000000000000002": {
      "balance": "456"
    }
  }
}
```

- 區塊鏈初始化，第一次開啟因為還沒有帳戶。命令中設定了節點 node1，所以執行成功後會產生名為 node1 的節點資料夾，包含 geth 和 keystore 兩個資料夾，keystore 負責存放所有帳號的密鑰。

```
geth --datadir "D:\BlockChain\node1" init "D:\BlockChain\genesis.json"
```

- 另一個 cmd 開啟錢包，設定帳戶和密碼。

```
geth attach ipc:\\.\pipe\geth.ipc
personal.newAccount("<password>")
```

- 重啟區塊鏈（最終使用版），要使用 "-allow-insecure-unlock"，才能解鎖帳

戶。

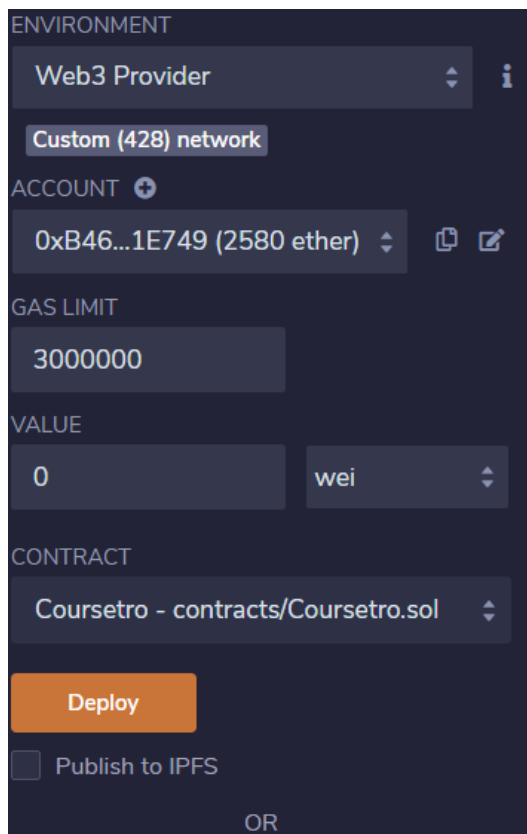
```
geth --identity "Node1" --http --ws --networkid 168 --nodiscover --maxpeers 5 --https.corsdomain "https://remix.ethereum.org" --http.api "personal,eth,net,web3" --http.port "8080" --datadir "D:\BlockChain\node1" --port "30303" --mine --cache=1024 --allow-insecure-unlock --unlock 0
```

註 1：開發過程中官方將 rpc 相關參數徹底進行淘汰，後續只能使用 http 指令，不然會無法進行連線。

註 2：出現下圖的錯誤。後來將參數--http.corsdomain 更改為 https 就能成功。

```
WARN [11-16|14:26:21.492] Served net_listening conn=127.0.0.1:53426 reqid=1845 t=0s err="the method net_listening does not exist/is not available"
```

- 開啟挖礦：`miner.start()`，一定要同步挖礦才能夠在鏈上進行交易。
- 連接 remix，將環境改為 Web3 Provider。
- 前面啟動節點時設定了 8080 port，於是將 localhost 的 port 號做更改。
- 重新 deploy 就會成功連接上，也可以看到對應的錢包位置與餘額。



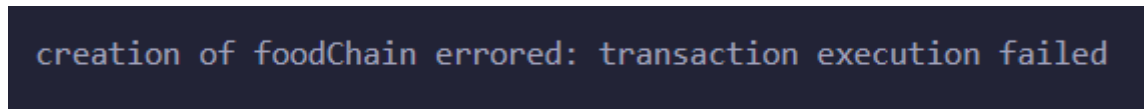
註 3：實際連線實測時如果抓不到 account list，就代表忘記將新註冊的帳號解鎖。

```
> web3.personal.unlockAccount(web3.personal.listAccounts[0], "Pass@w0rd", 15000)
true
> miner.start()
```

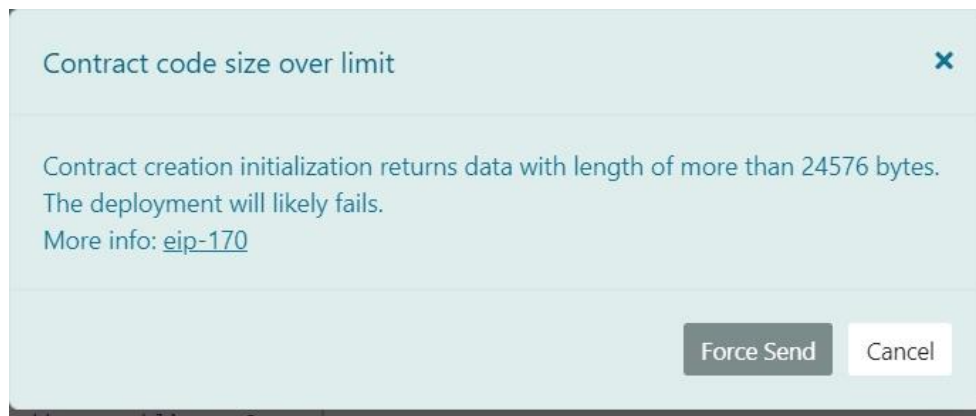
- 智慧合約說明

使用線上程式 remix IDE 編譯，Solidity 版本為 0.5.12。

1.1.0 後開發由於合約篇幅擴增、交易量變大，有交易的錯誤如下圖。後來將交易手續費（Gas Limit）增加改成 4000000。



1.1.1 的時候進一步出現了下圖的錯誤，code size 過大導致超出 remix 的限制，於是將合約內容進行整理，分割成兩份，分別是店家端的” storeALL” 與客戶端的” clientALL”。下面的說明皆以切割後的兩份合約為主。



版本的宣告：`pragma solidity >=0.5.12;`

工作區：`pragma experimental ABIEncoderV2;`

店家端合約（storeALL）

- 宣告全域 address 陣列 allStore[]，存放所有註冊店家的 address，方便 python 端資料取得。
- 兩張表 foodLists 跟 stores，foodList 在 store 裡以食材名稱為主鍵(如 beef001)mapping 宣告，為讓每個店家都可以有不只一份且獨立的食材表，資料形式為一(店家)對多(食材)也方便管理。

```
//食材表格
struct foodList{
    string id;
    address storeAddress;
    uint status; //啟用狀態
    uint inputTime; //食材進貨
    uint clearTime; //食材清洗
    bool isVaild;
}

//商店總表
struct store{
    address storeAddress;
    string private_key;
    string storeName;
    string account;
    string passWd;
    bool isVaild; //帳戶是否啟用
    bool isOpen;
    int i; //紀錄 foodList 筆數
    mapping(string => foodList) foodLists;
}
mapping(address => store)public stores;
```

- 另外兩張 `timeLists`、`allStoreInfos` 主要用於方便前端做快速資料取得與呈現。
- `timeList` 單純儲存各個食材的名稱、輸入與清洗時間，以店家 `address` 作為 mapping 主鍵，與總表不同的是資料內容以陣列儲存，意即 `foodID[“beef001”, “beef002” ...]` 此種形式。
- 而 `allStoreInfo` 主要儲存店家名稱、`address`、`icon` 也就是呈現在前端的頭像，圖片資料會儲存在 SQLite 裡，透過 `iconName` 的字串去呼叫，`isOpen` 為營業狀態、`isClear` 為清潔狀態。

```

//前端顧客查看店家列表
struct allStoreInfo{
    address storeAddress;
    string storeName;
    string iconName; //UI icon img Name
    bool isOpen;
    bool isClear;
}

struct timeList{ //用於前端呈現
    string[] foodID;
    uint[] inputTime;
    uint[] clearTime;
}

mapping(address => timeList) timeLists; mapping(address => allStoreInfo) public allStoreInfos;

```

- `onlyStore`：只有狀態為啟用的店家才可以執行的 `modifier` 函式。

```

//簡單的店家權限設置
modifier onlyStore(address _add){
    require(stores[_add].isVaild == true, "No permission!");
    _;
}

```

- 店家註冊功能：`setStore()`
- 對應表 `stores`，key `address`，店家註冊功能，關聯錢包所以目前只提供後臺人工註冊。`isVaild` 在註冊時即啟用，若未來該店家不再合作，則此欄位改為 `false`。
- 同步儲存表 `allStoreInfos`，key `address`，清潔狀態預設 `false`。

//店家註冊

```
function setStore(address _storeAddress, string memory _storeName,
    string memory _account, string memory _passWd, string memory _iconName) public{
    stores[_storeAddress].storeAddress = _storeAddress;
    // stores[_storeAddress].private_key = _private_key;
    stores[_storeAddress].storeName = _storeName;
    stores[_storeAddress].account = _account;
    stores[_storeAddress].passWd = _passWd;
    stores[_storeAddress].i = 0;
    stores[_storeAddress].isVaild = true;
    stores[_storeAddress].isOpen = false;

    allStoreInfos[_storeAddress].storeAddress = _storeAddress;
    allStoreInfos[_storeAddress].storeName = _storeName;
    allStoreInfos[_storeAddress].iconName = _iconName;
    allStoreInfos[_storeAddress].isClear = false;

    allStore.push(_storeAddress); //紀錄全部店家之address
}
```

- 店家登入功能：checkStore()
- 前端儲存的 address(第一次登入後儲存於 local Storage)，手動輸入的帳號加密碼進行驗證，進行 encode 和轉碼才能判定字串相等，回傳 true/false。

//店家登入判斷

```
function checkStore(address _storeAddress, string memory _account,
    string memory _passWd) public view returns(bool){
    if(keccak256(abi.encodePacked(stores[_storeAddress].account)) == keccak256(abi.encodePacked(_account))){
        if(keccak256(abi.encodePacked(stores[_storeAddress].passWd)) == keccak256(abi.encodePacked(_passWd))){
            return true;
        }
    }else{
        return false;
    }
}
```

- 店家營業狀態更改/取得：setStoreOpen()、setStoreClose()、getStoreState()
- setStoreOpen()、setStoreClose() 透過前端的按鈕針對 stores、allStoreInfos 兩張表的參數狀態進行更改，套用 modifier 對匯入帳號進行權限設定。
- getStoreState() 則是單純取得營業狀態呈現於前端


```

//開始營業
function setStoreOpen(address _storeAddress) public onlyStore(_storeAddress){
    stores[_storeAddress].isOpen = true;
    allStoreInfos[_storeAddress].isOpen = true;
}
//結束營業
function setStoreClose(address _storeAddress) public onlyStore(_storeAddress){
    stores[_storeAddress].isOpen = false;
    allStoreInfos[_storeAddress].isOpen = false;
}
//取得營業狀態
function getStoreState(address _storeAddress) public view returns(bool){
    return(stores[_storeAddress].isOpen);
}

```

- 食材進貨：setDeliverTime()
- 同樣 onlyStore，並且該食材不能已被啟用(require)，清洗時間預設為 0，`foodLists` 綁定於店家總表內，每新增一筆則總表內的食材筆數加一，方便統整計算。
- 同步也將資料寫入 `timeLists` 表中。

```

//食材進貨輸入
function setDeliverTime(string memory _id, address _storeAddress) public onlyStore(_storeAddress){
    require(stores[_storeAddress].foodLists[_id].isValid != true);
    stores[_storeAddress].foodLists[_id] = foodList({
        id: _id,
        storeAddress: _storeAddress,
        status: 1,
        inputTime: block.timestamp,
        clearTime: 0,
        isValid: true
    });
    stores[_storeAddress].i = stores[_storeAddress].i+1;

    //將時間與食材資料填入 timeLists
    timeLists[_storeAddress].foodID.push(_id);
    timeLists[_storeAddress].inputTime.push(stores[_storeAddress].foodLists[_id].inputTime);
}

```

- 取得單筆進貨時間/所有食材名稱/所有進貨時間：getDeliverTime()、getAllFoodID()、getAllDeliverTime()

```

//取得單筆食材進貨時間
function getDeliverTime(string memory _id, address _storeAddress) public view returns(uint){
    return (stores[_storeAddress].foodLists[_id].inputTime);
}
//取得該店家所有登錄的食材名稱
function getAllFoodID(address _storeAddress) public view returns(string[] memory){
    return(timeLists[_storeAddress].foodID);
}
//取得該店家所有登錄的食材進貨時間
function getAllDeliverTime(address _storeAddress) public view returns(uint[] memory){
    return(timeLists[_storeAddress].inputTime);
}

```

- 登錄食材清洗時間：setFoodCleanTime()
- onlyStore
- 兩個 require，分別滿足該食材存在且進貨（state=1）、食材未清洗過的條件（一個食材有進貨/清洗/用盡三狀態，一次只存在一種狀態每種狀態不可回溯及重複）
- stores 中對應 foodList 的 clearTime 直接更新
- timeLists 中由於清洗時間以 array 存在，故每次更新則重新定義其 clearTime[]，先做刪除再重新代入 foodList 的資料，流程為透過 address 取得總表（黃）裡的每個食材（for 迴圈 j）表（藍）的清洗時間，食材表的主鍵透過 timeList 的 foodID[j]（橘）獲得

```
//登錄食材清洗時間
function setFoodCleanTime(string memory _id, address _storeAddress) public onlyStore(_storeAddress){
    require(stores[_storeAddress].foodLists[_id].status == 1); //已有食材進貨紀錄
    require(stores[_storeAddress].foodLists[_id].clearTime == 0); //未曾登入過食材清洗
    stores[_storeAddress].foodLists[_id].clearTime = block.timestamp;

    //重新排列 timeList中食材清洗時間
    delete timeLists[_storeAddress].clearTime;
    for(uint j=0;j<timeLists[_storeAddress].foodID.length;j++){
        timeLists[_storeAddress].clearTime.push(stores[_storeAddress].foodLists[timeLists[_storeAddress].foodID[j]].clearTime);
    }
}
```

- 取得單筆食材狀態/單筆食材清洗時間/所有食材清洗時間：
getFoodState()、getFoodCleanTime()、getAllCleanTime()

```
//取得單筆食材狀態
function getFoodState(string memory _id, address _storeAddress) public view returns(uint){
    return (stores[_storeAddress].foodLists[_id].status);
}
//取得單筆食材清洗時間
function getFoodCleanTime(string memory _id, address _storeAddress) public view returns(uint){
    return (stores[_storeAddress].foodLists[_id].clearTime);
}
//取得該店家所有登錄的食材的清洗時間
function getAllCleanTime(address _storeAddress) public view returns(uint[] memory){
    return(timeLists[_storeAddress].clearTime);
}
```

- 刪除食材紀錄：deleteFood()
- onlyStore
- require 要該食材存在(state=1)
- stores 中對應 foodList 直接刪除，並且將總表中代表食材數的 i 欄位-1。
- 針對 timeList 表，有同樣問題，由於資料存在模式為非指向型陣列，於是先使用 for 迴圈當判定到對應的 foodID 時將對應位置的 inputTime 與 clearTime 數值刪除，並且透過參數 delT 紀錄刪除的陣列位置。
- 由於刪除後該位置仍存在，意即後面的數值不會自動往前替補。故需再透過 for 迴圈，自前面紀錄的 delT 位置開始，每個欄位往前移動
- 最後再手動將三個陣列的 length-1（最後面的空欄位會被去掉）

```

//刪除食材紀錄
function deleteFood(string memory _id, address _storeAddress) public onlyStore(_storeAddress){
    require(stores[_storeAddress].foodLists[_id].status == 1);
    delete stores[_storeAddress].foodLists[_id];
    stores[_storeAddress].i = stores[_storeAddress].i-1;

    //將指定的食材資料在timeList的三個array中移除
    uint delT;
    for(uint j=0;j<timeLists[_storeAddress].foodID.length;j++){
        if(keccak256(abi.encodePacked(timeLists[_storeAddress].foodID[j]))
            == keccak256(abi.encodePacked(_id))){
            delete timeLists[_storeAddress].foodID[j];
            delete timeLists[_storeAddress].inputTime[j];
            delete timeLists[_storeAddress].clearTime[j];

            delT = j; //紀錄刪除點
            break;
        }
    }
    //重新放置 array 位置(將刪除的index覆蓋掉)
    for(uint j=delT;j<timeLists[_storeAddress].foodID.length-1;j++){
        timeLists[_storeAddress].foodID[j] = timeLists[_storeAddress].foodID[j+1];
        timeLists[_storeAddress].inputTime[j] = timeLists[_storeAddress].inputTime[j+1];
        timeLists[_storeAddress].clearTime[j] = timeLists[_storeAddress].clearTime[j+1];
    }
    timeLists[_storeAddress].foodID.length--;
    timeLists[_storeAddress].inputTime.length--;
    timeLists[_storeAddress].clearTime.length--;
}

```

- 環境清洗功能使用 `locationTimes` 表紀錄，mapping 的主鍵同樣為 `address`，每個店家只會有一張 `locationTime`，

```

//環境清理時間表
struct locationTime{
    address storeAddress;
    uint currentTime;
    uint futureTime;
    bool status;
}
mapping(address => locationTime) public locationTimes;

```

- 紀錄/取得店家環境清洗時間：`setLocationTime()`、`getLocationTime()`
- 登錄功能 `onlyStore`
- 更新 `locationTimes` 表，自動在紀錄該次清洗時間時預設一個下次更新狀態時間，主要對應下一個 `refresh()` 功能，前端的 `allStoreInfos` 本日是否清洗狀態改為 `true`。

```

//登錄店家環境清洗時間
function setLocationTime(address _storeAddress) public onlyStore(_storeAddress){
    locationTimes[_storeAddress] = locationTime({
        storeAddress: _storeAddress,
        currentTime: block.timestamp,
        futureTime: block.timestamp+86400,
        status: true
    });

    allStoreInfos[_storeAddress].isClear = true;
}

//取得該店家之環境清洗時間
function getLocationTime(address _storeAddress) public view returns(uint){
    return (locationTimes[_storeAddress].currentTime);
}

```

- 店家每日登入的狀態更新：refresh()
- onlyStore
- require 當前時間需大於上次清洗的時間一天後才可刷新（防止重複登入）
- 更新 locationTimes 的 currentTime、state 欄位為 false

```
//前端登入畫面中"繼續"呼叫的重製方法
function refresh(address _storeAddress) public onlyStore(_storeAddress){
    require(block.timestamp >= locationTimes[_storeAddress].futureTime);
    locationTimes[_storeAddress].currentTime = block.timestamp;
    locationTimes[_storeAddress].status = false;
}
```

- 取得清洗狀態：getLocationState()

```
function getLocationStatus(address _storeAddress) public view returns(bool){
    return (locationTimes[_storeAddress].status);
}
```

- 取得所有店家地址/取得地址對應的總表：getAllStore()、storeInfoForUser()

```
//取得所有店家地址
function getAllStore() public view returns(address[] memory){
    return(allStore);
}

//前端py透過上一個方法取得的店家address寫一個迴圈
function storeInfoForUser(address _storeAddress) public view returns(allStoreInfo memory){
    return(allStoreInfos[_storeAddress]);
}
```

客戶端合約 (clientALL)

- 宣告全域 address 陣列 allAccount[]，存放所有註冊店家的 address，方便 python 端資料取得。
- user 儲存用戶的錢包地址、用戶名稱、帳號、密碼，以帳號字串為主鍵 mapping 表 users

```
//使用者
struct user{
    address payable wallet;
    string account;
    string userName;
    string passWd;
}
mapping(string => user) public users;
```

- 顧客註冊功能：setUser()
- 前端即可註冊成為一般顧客，由 python 端為用戶建立錢包地址，同步將帳號儲存至 allAccount[]

```
//使用者註冊
function setUser(address payable _wallet, string memory _userName,
    string memory _account, string memory _passwd) public{
    users[_account] = user(_wallet, _userName, _account, _passwd);

    allAccount.push(_account); //紀錄全部使用者帳號
}

```

- 顧客登入功能：checkUser()

```
//使用者登入
function checkUser(string memory _account, string memory _passwd)
    public view returns(bool){
    if(keccak256(abi.encodePacked(users[_account].passwd)) ==
        keccak256(abi.encodePacked(_passwd))){
        return true;
    }else{
        return false;
    }
}

```

- 取得所有使用者帳號(前端)：getAllAccount()

```
//取得所有使用者帳號
function getAllAccount() public view returns(string[] memory){
    return(allAccount);
}

```

- (未啟用)allComments 評論總表

```
//用戶評論
struct allComment{
    uint id;
    address storeAddress;
    string storeName;
    string[] userName;
    string[] comment;
}
mapping(address => allComment)public allComments;

```

- (未啟用)新增評論與取得功能：setComment()、getComment()

```
//匯入評論
function setComment(address _storeAddress, string memory _storeName,
    string memory _userName, string memory _comment) public{
    allComments[_storeAddress].storeAddress = _storeAddress;
    allComments[_storeAddress].storeName = _storeName;
    allComments[_storeAddress].id++;
    allComments[_storeAddress].userName.push(_userName);
    allComments[_storeAddress].comment.push(_comment);
}

//取得評論
function getComment(address _storeAddress) public view returns(uint, string[] memory, string[] memory){
    return(allComments[_storeAddress].id, allComments[_storeAddress].userName, allComments[_storeAddress].comment);
}

```

亞大幣(AsiaToken)

- 使用 ERC20 標準，基本只更改發行的貨幣內容。

執行流程

- 首頁

- App 的初始頁面 (index.html)。
- 在此選擇登入為店家/顧客。



- 店家端功能



- choose_store.html
- 當前不開放在前端註冊店家，需透過後臺綁定錢包。
- 點擊登入 > [choose_store_conn.js](#)：判斷本機是否儲存過錢包地址資料，將登入引導至不同頁面。

```
console.log(localStorage.address);
function check(){
  if(localStorage.address==undefined){
    window.location.href='firststone-login.html';
  }
  else{
    window.location.href='store-login.html';
  }
};
```

- 右上角的菜單打開可以重新選擇。
- 快速切換至顧客註冊/顧客登入畫面。

- 第一次登入：firststone-login.html
- **firststone-login.js**：onload()先建立連線，點擊按鈕”Enter”後呼叫check()傳送欄位資料。
- py 中透過 type=firstLogin 連線 Contract.py，合約 **checkStore()**
- 店家首次登入需手動輸入錢包地址作為綁定，登入成功後 address 就會記錄在 localStorage 中。

```
console.log(localStorage.address);
function check(){
  if(localStorage.address==undefined){
    window.location.href='firststone-login.html';
  }
  else{
    window.location.href='store-login.html';
  }
};
```

- 之後的登入：store-login.html
- **store-login.js**，其他連線和第一次登入大同小異。
- onload()檢查本地 localStorage 中 address 欄位是否存在，並將 address 資料傳送到後端獲得對應的店家名稱，並呈現在前端畫面上。
- “reset” 按鈕可以重設店家的錢包，狀態會回歸到第一次登入畫面。

第一次登入

之後

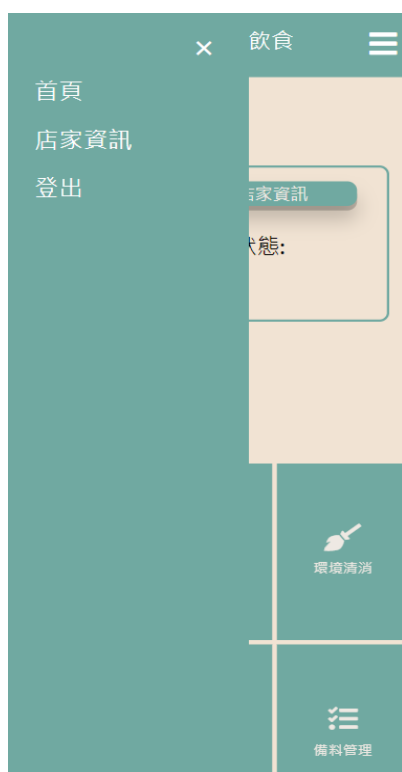
- 登入成功畫面：loginafter.html



- `loginafter.js`：取得當前時間呈現
- `loginafter_conn.js`：`onload()`裡將圖片、店家名稱都丟回頁面。並且建立連線傳送 `type=refresh` 到 py，連線 `Contract.py`，合約 `refresh()`



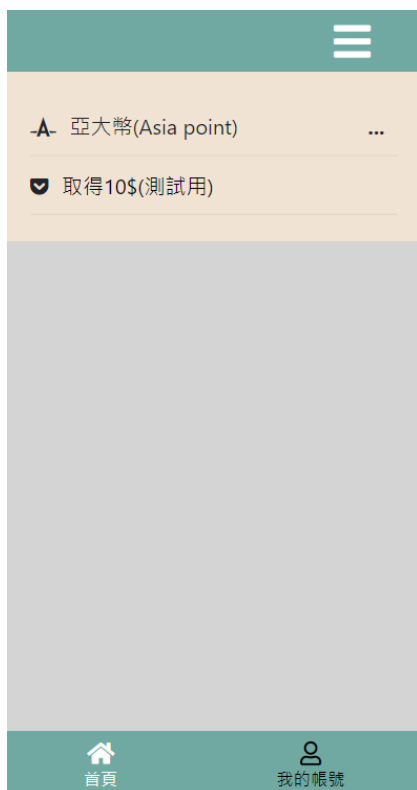
- 店家首頁：`main.html`
- `main_conn.js`
- `onload()`裡更新畫面上的營業狀態 (`localStorage` 的 `state`)，傳送 `type=setTime`。
- 此按鈕為該店家的 Address 地址，以 QR code 形式呈現。
- 可跳轉店家資訊 `store-info.html` 查看亞大幣的餘額。
- 左邊區域為商家所提供的 logo。
- 食材進貨 `setDeliverTime()`、食材清洗 `setFoodCleanTime()`、環境清消 `setlocationTime()`、食材耗盡 `deleteFood()`：此按鈕點選可開啟 QR code 掃描器，掃描完成透過 js 的 `scan()` 新增，可在表單中 (`foodin-list.html`) 看見新增項目。
- 營業狀態：點選按鈕進入 (`business-status.html`) 開關按鈕更改營業狀態。
- 備料管理：點選按鈕進入 (`foodin-list.html`) 可查詢目前所有清消及食材資訊。



- 導航欄按鈕：分別為首頁 (`main.html`)、店家資訊 (`store-info.html`)、登出 (`index.html`)



- 店家營業狀態：`business-status.html`
- `business-status_conn.js`：`onload()`裡預設先透過 `localStorage` 資料更新，再連線後端傳送 `type=StoreStateLoad`，合約 `getStoreState()`取得區塊鏈中欄位狀態。
- 此頁有兩個按鈕，分別為營業中、打烊，店家可透過此來更改現在店家狀態，顧客端可透過首頁來觀看該店家的營業狀態。
- 切換營業中呼叫 `js` 的 `opening()`，傳送 `type=StoreStateOpen`，合約 `setStoreOpen()`
- 切換已打烊呼叫 `js` 的 `closed()`，傳送 `type=StoreStateClose`，合約 `setStoreClose()`
- 初始預設為沒有任何狀態。



- 店家資訊：`store-info.html`
- 亞大幣欄位會顯示該店家錢包 `address` 內的餘額。
- `store-info.js`：`onload()`中傳送 `type=getBalance`，呼叫 `asiaToken` 合約 `balanceOf()`，取得餘額後顯示於頁面上。
- ```
setTimeout(getbalance(),3000);
```

 三秒刷新一次。
- 在測試階段設計了取得\$10的按鈕，會呼叫 `getCoin()`，傳送 `type=transfer`，後端呼叫合約 `transfer()`的函式，由主要帳戶向本帳戶發送\$10亞大測試幣。
-

- 備料管理功能
- 以食材進貨管理 (foodin-list.html) 為首頁，主要呈現當前已進貨啟用的食材，都有食材、日期時間、狀態三個欄位。
- **foodin-list\_conn.js**: **onload()** 中先設定下方列更新時間，再傳送 **type=DeliverTime**，呼叫合約的 **getAllDeliverTime()** 回傳資料，由於資料不同  

```
//從python傳進來的值會長 --> [1630734778, 1630734813]
//放入datas_arr裡後 --> [['1630734778, 1630734813']]
```

，在 js 的第 45~63 行進行 for 迴圈的字串處理。
- 第 67~108 行則是將資料以表格形式在頁面上呈現，其中 if datas 的長度==3  

```
check
[['1630734778, 1630734813']]
[['Beef001, Beef002']]
```

的判斷主要依此判定。
- 功能列第二個按鈕可以切換食材清洗管理頁 (food-list.html)，主要呈現當前已進貨啟用的食材清洗狀態。
- **food-list\_conn.js**: **onload()** 中同樣更新時間，再傳送 **type=CleanTime**，呼叫合約的 **getAllCleanTime()** 回傳資料。
- 第三個按鈕可以切換環境清消管理頁 (kitchen-list.html)，主要呈現該店家的上次清洗紀錄及當前清洗狀態。
- **kitchen-list\_conn.js**: **onload()** 中同樣更新時間，再傳送 **type=LocationTime**，呼叫合約的 **getlocationTime()** 回傳資料。
- 三頁的功能與程式處理大同小異。

| 食材進貨管理            |                      |    |
|-------------------|----------------------|----|
| 食材                | 日期時間                 | 狀態 |
| Beef001           | 2021/9/20 上午10:17:47 | 進貨 |
| Beef003           | 2021/9/20 上午10:18:06 | 進貨 |
| Beef004           | 2021/9/20 上午10:18:15 | 進貨 |
| Beef005           | 2021/9/20 上午10:18:54 | 進貨 |
| Beef006           | 2021/9/20 上午10:19:06 | 進貨 |
| 更新時間: 09-20 10:19 |                      |    |

| 食材清洗管理            |                      |     |
|-------------------|----------------------|-----|
| 食材                | 日期時間                 | 狀態  |
| Beef001           | 2021/9/20 上午10:18:01 | 清洗  |
| Beef003           | 2021/9/20 上午10:18:24 | 清洗  |
| Beef004           | 0                    | 未清洗 |
| Beef005           | 0                    | 未清洗 |
| Beef006           | 2021/9/20 上午10:19:15 | 清洗  |
| 更新時間: 09-20 10:20 |                      |     |

| 環境清消管理               |     |
|----------------------|-----|
| 時間                   | 狀態  |
| 2021/9/20 上午10:20:32 | 未清理 |
| 更新時間: 09-20 10:20    |     |

- 顧客端功能

- 顧客註冊/登入：choose\_custormer.html -> customer-login.html/  
signup.html



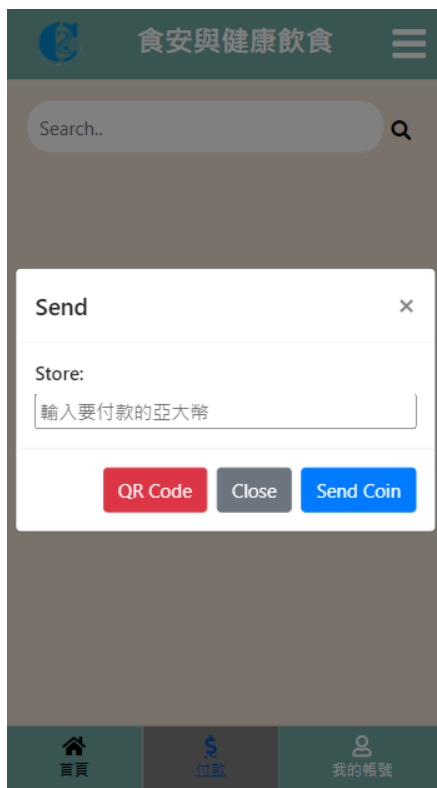
- 可以選擇用已有帳號登入或是進行註冊，會在後端幫該客戶建立一個錢包帳號。
- 登入功能 `customer-login_connjs`：`onload()` 中傳送 `type=customerLogin`，在點擊 submit 後才透過 `check()` 將帳號和密碼資料傳送，呼叫合約的 `checkUser()` 進行驗證，同時後端會透過 `getUserName()` 取得用戶的暱稱與 address 回傳前端，將資料放入 `localStorage` 中。
- 註冊功能 `signup.js` 中由按鈕 submit 觸發 `getFlag()`，和第二次的密碼輸入時都會呼叫 `checkText()` 檢查兩次密碼是否輸入正確，基本上在輸入第二次時若不正確該欄位的外框就會變成紅色。
- 而 `signup_conn.js`：點擊按鈕 submit 觸發 `check()` 中傳送 `type=signup` 與帳號密碼，在後端先呼叫 `getAllAccount()` 對比此帳號是否存在，若存在則註冊失敗，不存在則透過 "Contract.py" 中 `createWallet()` 創建錢包。

```
def createWallet(self, account):
 newAccount = self.w3.eth.personal.new_account(account)
 # newAddress = self.w3.eth.personal.list_accounts()
 print('address is : {}'.format(newAccount))
 return newAccount
```

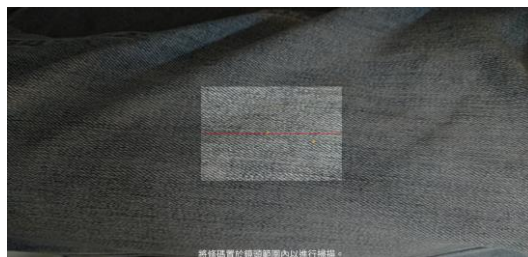


- 顧客主頁：customer-home.html
- customer-home\_conn.js：onload()中傳送 type=AllStore，呼叫合約的 storeInfoForUser() 抓取所有店家資料，在 onclose 確定有收到資料後呼叫 setInfo()將每個店家的資料如左圖呈現。
- 上方搜尋欄透過 runSearch()快速搜尋店家，店名、清洗狀態、營業狀態都可以做篩選。
- 內容包含店家註冊所提供 Logo，以及店名、環境清洗狀態、營業時間。點選店家可跳轉至店家資訊 (customer-storeInfo.html)，可查詢食安情況，以及菜單內容。
- 我的帳號按鈕可以跳轉到亞太幣餘額頁 (customer-info.html)。
- 付款按鈕會跳出支付畫面，在 html 第 59~80 行，可以掃描店家錢包地址(Address)，輸入付款金額後送出，店家端馬上就能入帳(Asia Token)

#### 付款交易功能



- QR Code 按鈕：點選系統呼叫 barcodeScanner API，首次使用需同意開啟鏡頭權限，開啟後就可掃描收款方的錢包地址。
- Store：抓取掃描後匹配的店家名稱。(下圖為掃描畫面)
- Close：關閉付款懸浮視窗。



- Send Coin：在輸入欄裡輸入付款金額，按下 Send Coin 使用 transfer()函式傳送 type=transferFrom，呼叫合約的 approve()然後 transferFrom()，將指定的金額自顧客錢包傳送到目標店家，便能付款成功，未輸入金額則會跳出 Alert 錯誤訊息。

- 前一頁點擊(customer\_home.html)點擊該店家時從 words[i][0] 抓到的 address 並把它存進網址，在本頁的 js 檔取得該 address。
- 店家資訊：customer\_storeInfo.html
- Customer-storeInfo\_conn.js：onload() 中傳送 type=storeInfoForUser，呼叫合約 storeInfoForUser()，取得資料後呼叫 setInfo() 函式呈現。

畫面



```
function setInfo(){
 for(let i=0;i<words.length;i++){
 let tmp = "<button id='store'+i+' ' class='store' onclick='javascript:window.location.href = \""+customer_storeInfo.html?\" + words[i][0] + \"'>";
 "div id='photo' class='photo'>";
 tmp += "</div>";
 tmp += "<div id='information' class='information'><h3>"+words[i][1]+</h3>";
 if(words[i][4]==true) tmp += "<p>環境狀態: 已清潔</p>";
 else tmp += "<p>環境狀態: 未更新</p>";
 if(words[i][3]==true) tmp += "<p>營業時間: 營業中</p></div></button>";
 else tmp += "<p>營業時間: 休息中</p></div></button>";
 $('#stores').append(tmp);
 }
}
```

server

```
customer-storeInfo頁面
elif n=="9":
 await websocket.send("check")
 address = await websocket.recv()

 sum = contract.storeInfoForUser(address)
 await websocket.send(str(sum))
```