

IPFS Camera

IPFS + Camera + BlockChain



Code block 들을 기반으로 IPFS Camera 의 전반적인 설명을 첨부.

Reporter Name

이 봉 호

목차

-개요

- 사용된 Device
- Device 별 역할
- 간단한 도식화
- 관련된 구현 기능

-사용된 소프트웨어 설명

- IPFS 주요 명령어
- openRTSP 주요 명령어
- geth 주요 명령어
- sqlite

-Code

- 기능별 script
- 기능별 code

-그 외 issue

- IPFS 관련
- geth 관련

-참고 사이트

-개요

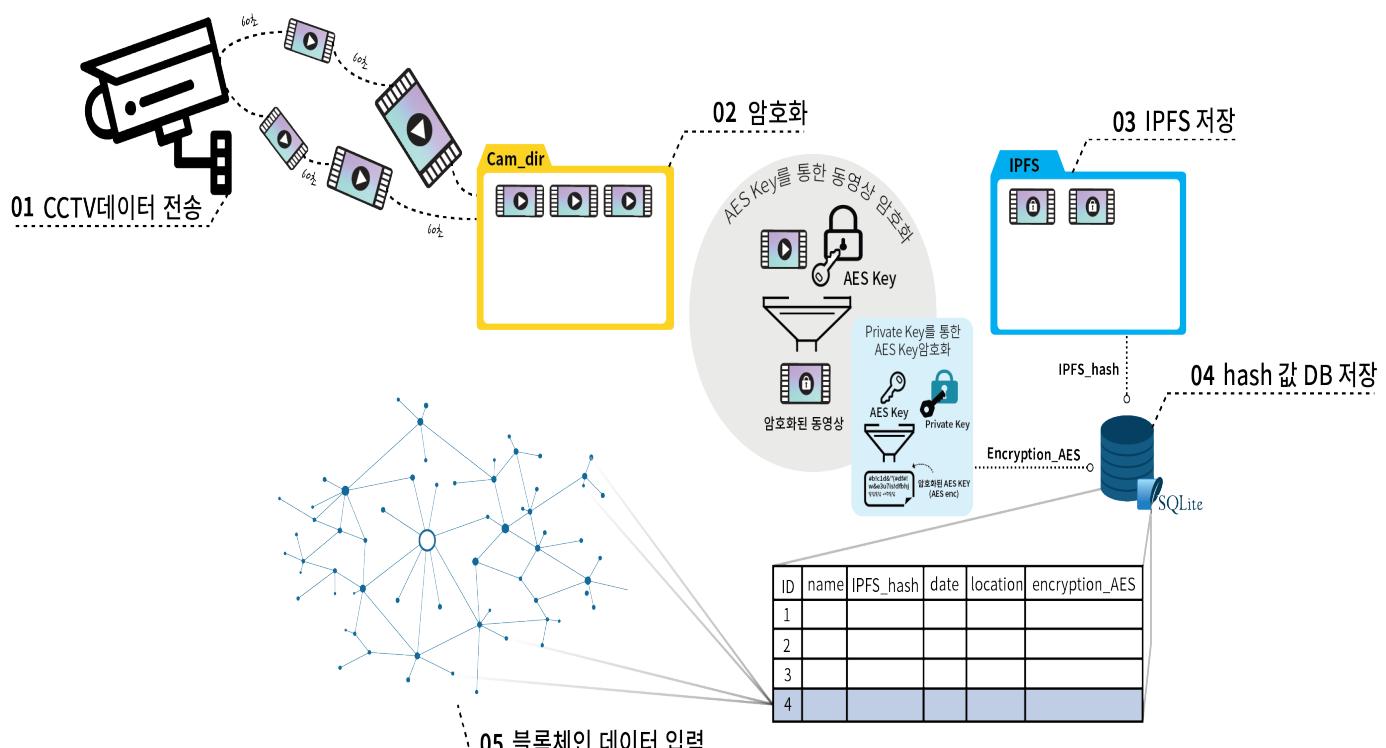
IPFS Camera 의 back-end 에 해당하는 프로그램으로써 Programming Language 로는 Python3.x 를 사용했습니다.

프로그램의 테스트를 위해 사용한 Device 로는 IP Camera, Raspberry pi3, Mac Mini 가 있습니다.

IP Camera 는 영상의 저장(테스트는 1 분 및 1 분 30 초 단위의 clip 을 사용)을 위해 사용하였고 Raspberry pi3 는 IPFS Camera 의 node 로써 IP Camera 로부터 영상을 받아와서 일련의 작업 (영상의 복호화 및 메타데이터화, IPFS 저장 등)을 수행합니다.

Mac mini 는 현재 블록체인의 노드(Geth node)로써 각 Raspberry pi3 의 영상 메타데이터를 smart contract 에 저장하기 위해 필요한 synced chain-data 를 제공합니다.

즉 Raspberry pi 각각마다 chain-data 의 synchronization 가 필요 없이 Mac Mini(BlockChain-node)의 chain-data 를 사용하여 smart contract 와 통신합니다.



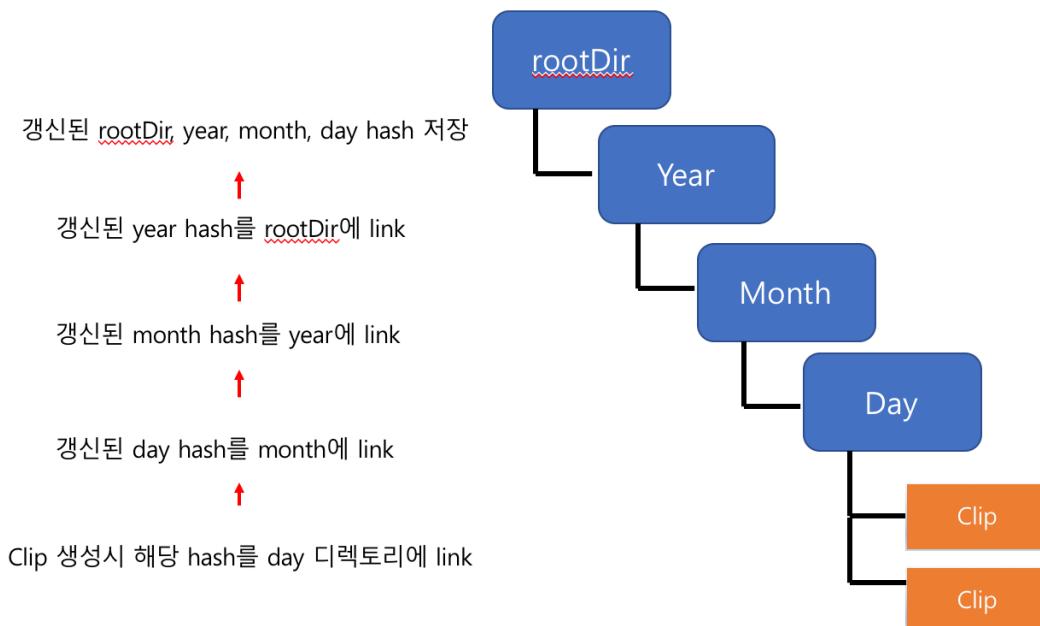
위의 주요 사항 외에도 몇 가지 관련된 구현 기능들이 존재합니다. ‘**Camera Merkle directory(IPFS directory 구조화)**’ 및 ‘**Smart Contract Data 의 Database 저장**’이 그 예입니다.

먼저 **Camera Merkle dircetory(이하 머클 디렉토리)**는 IPFS 에서 제공하는 명령어 옵션을 이용하여 Python, sqlite 에 적용하여 구현하였습니다.

머클 디렉토리는 IPFS Camera 의 영상 clip 을 smart contract 에 upload 하는데에 소비되는 gas 비용(수수료)를 비약적으로 절약하는 방법으로써 clip 하나하나가 아닌 clip 이 저장된 directory 의 메타데이터를 smart contract 에 저장하는 방향을 염두에 두고 개발하였습니다.

머클 디렉토리의 구조는 rootDir(각 카메라의 ID)를 root node 로써 가지고 하위에 year(년도) directory, 그 밑에 month(월) directory 를 갖습니다. 그리고 말단 노드는 각 영상의 clip 이 저장됩니다.

‘**Smart Contract Data 의 Database 저장**’(이하 Contract To DB)의 경우 이후 개발에 있어서 영상의 메타데이터 필요시 Smart contract 에 직접 접근하지 않고 서버의 DB 에 접근할 수 있게 해줍니다. Smart contract 에 저장된 metaData 의 경우, 각 컬럼들이 구분되지 않은 String 형식이기 때문에 전처리 과정이 필요하지만, DataBase 에 저장할 때에 이러한 전처리를 모두 거치고 저장하였기 때문에 더욱 효율적인 개발이 가능할 것으로 예상됩니다.



<머클 디렉토리의 도식화>

-사용된 소프트웨어 설명

1. IPFS 주요 명령어

IPFS 에는 다양한 [명령어](#)들이 존재합니다. 여기서는 프로그램에 사용된 명령어들만 정리하겠습니다. 보다 자세한 내용은 공식 웹페이지를 참조하세요.

1] ipfs add <파일명> : 단일 파일에 대하여 저장하기 위한 명령어 입니다. 반환값으로는 <Content hash>가 주어집니다.

```
Lees-MacBook:~ leebongho$ ipfs add hello.txt
added QmbFMke1KXqnYyBBWxB74N4c5SBnJMVAiMNRCGu6x1AwQH hello.txt
Lees-MacBook:~ leebongho$
```

2] ipfs add -r <디렉토리명> : 파일 및 디렉토리를 포함한 디렉토리를 저장하기 위한 명령어 입니다. 반환값으로는 포함된 파일 및 디렉토리 까지 <Content Hash>를 반환합니다.

```
Lees-MacBook:~ leebongho$ ipfs add -r rootDir
added QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn rootDir/2018/1
added QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn rootDir/2018/2
added QmY4qm8tq5XukFe5WYRH3qgtwn3AMtNW9ecpX2BzGpAMhx rootDir/2018
added QmWzJAY8LhDNTqmEbyKKL7auNpnWNMFqC65aRRtaGFYC7i rootDir
```

3] ipfs get <hash> : IPFS hash 에 대해서 get 요청을 수행하는 명령어 입니다.

4] ipfs object patch <root hash> add-link <file or directory name> <file or directory hash> : 기존의 IPFS 에 저장된 디렉토리에 다른 파일 및 디렉토리를 link 하기 위한 명령어 입니다. 해당 명령어를 사용해서 머클 디렉토리를 구현했습니다.

```
added QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn rootDir/2018/1
added QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn rootDir/2018/2
added QmY4qm8tq5XukFe5WYRH3qgtwn3AMtNW9ecpX2BzGpAMhx rootDir/2018
added QmWzJAY8LhDNTqmEbyKKL7auNpnWNMFqC65aRRtaGFYC7i rootDir
Lees-MacBook:~ leebongho$ ipfs object patch QmWzJAY8LhDNTqmEbyKKL7auNpnWNMFqC65aRRtaGFYC7i add-link hello.txt QmbFM
ke1KXqnYyBBWxB74N4c5SBnJMVAiMNRCGu6x1AwQH
QmWBtAZGpZ8p5Zukai49cRNY8Kco5jZYW4oYFdKZeRL1Vg
```

위의 명령어는 기존의 rootDir 의 디렉토리에 hello.txt 를 link 시켜주었습니다. 즉, rootDir 하위에 hello.txt 를 넣어준 셈이 됩니다. 붉은 박스로 표시한 hash 는 link 를 시켜줌에 따라 변경된 rootDir 의 hash 입니다.

5] ipfs daemon : ipfs 의 daemon 을 구동하는 명령어입니다. 즉 IPFS p2p network 에 참여하기 위한 명령어입니다. daemon 이 구동되어 있지 않으면 외부 노드가 가지고 있는 ipfs file 에 대해서 get 요청할 수 없습니다.

```
Lees-MacBook:~ leebongho$ ipfs daemon
Initializing daemon...
Adjusting current ulimit to 2048...
Successfully raised file descriptor limit to 2048.
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip4/192.168.0.69/tcp/4001
Swarm listening on /ip6/:1/tcp/4001
Swarm listening on /p2p-circuit/ipfs/QmbYUbuESze12tPCQjhckYyL6j5B8NKHdViCF5xVxsKYjg
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/192.168.0.69/tcp/4001
Swarm announcing /ip6/:1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
```

6] 간혹 여러가지 이유로 ipfs api not running 에러를 만날 수 있습니다.

해당 에러가 발생하면 add 및 get 등 모든 ipfs 명령어를 사용할 수 없습니다. ‘ipfs repo fsck’ 명령을 입력해 주신 뒤에 다시 daemon 을 구동하면 해결됩니다. 해당 에러에 대한 이유는 아직 명확하지 않습니다. 포럼에서도 ‘bug’ 라고 표현을 합니다. ipfs repo 에 lock file 이 생성되어 ipfs account 가 lock 이 걸리기 때문에 해당 에러가 발생하므로 lock file 을 제거하는 명령어인 ‘ipfs repo fsck’를 입력 해주시면 됩니다. ipfs daemon 은 별도로 다시 실행시켜주셔야합니다.

2. openRTSP 주요 명령어

openRTSP 의 경우 IP Camera 의 영상 스트림을 RTSP 프로토콜을 사용하여 요청할 수 있는 프로그램입니다. IPFS 와 마찬가지로 console 명령어를 이용합니다.

먼저 프로그램에 사용된 명령어 옵션입니다.

```
openRTSP -D 3 -B 250000 -b 250000 -c -i -F aCAM -P 90 -u admin admin rtsp://192.168.1.26/11
```

위의 명령어는 192.168.1.26 주소를 갖고 ID:admin, Password:admin 인 IP Camera 의 영상을 90 초 마다 aCAM000_000 이라는 이름으로 저장하겠다는 명령입니다.

위처럼 저장될 영상의 이름을 지정하거나 나눌 시간을 지정할 수 있습니다.

명령어 옵션에 대한 자세한 사항은 [이곳](#)에서 확인하세요.

3. Geth 주요 명령어

geth 를 이용해서 chain data 를 동기화 하는데에는 많은 옵션들이 있습니다.

우선 저희는 main network 를 이용하지 않고 testnet / private net 을 이용합니다.

testnet 은 이름 그대로 smart contract 관련 테스트를 하기 위해 다른 외부 노드들과 연결되는 환경이고 private 은 외부 노드들과 연결되지 않은, genesis block 부터 difficulty 까지 직접 설정하는 환경입니다.

testnet :

```
geth --rpc --rpccorsdomain "*" --port "30303" --rpcapi "db,eth,net,web3,personal,admin,debug,txpool" --testnet
```

위의 명령어가 console 상에서 이더리움 testnet(ropsten) 동기화 명령어 입니다. 명령어 옵션은 다양합니다. 위의 명령어는 제가 실행하는 내용이고 필요에 따라 옵션을 추가하실 수 있습니다. 보다 다양한 옵션을 확인하시려면 [이곳](#)을 참고하세요. 명령어를 살펴보면 다른 옵션보다도 중요한것은 --rpcapi 이하의 “db, eth…”이

부분입니다. 해당 옵션(api)들을 설정해 주셔야 이후에 web3.js 또는 web.py 를 이용해서 geth 의 chain data 를 사용하시는데 문제가 발생하지 않습니다.

Private net : geth --datadir chain-data --rpc --rpcport "8545" --rpcaddr "0.0.0.0" --rpccorsdomain "*" --port "30303" --mine --rpcapi "db,eth,net,web3,personal,admin,debug,txpool" --networkid 920410

위의 명령어는 private net 동기화 명령어입니다. Testnet 과 다른부분이 몇 가지 눈에 띄실겁니다. 먼저 rpcport 를 지정하고 rpcaddr 역시 지정해준 뒤 -mine 옵션으로 자체 채굴을 시작합니다. Rpcport 및 rpcaddr 명령어 옵션이 추가되어야 다른 디바이스에서 chain-data 에 접근하여 사용할 수 있습니다.

4. sqlite

sqlite 의 명령어는 여타 다른 데이터베이스의 명령어와 같습니다. Select, insert, update, delete, 등으로 조작이 가능합니다.

Database 접속 : sqlite3 <database 파일 이름>

```
ibonghoui-MacBook-Pro:monitoring leebongho$ sqlite3 test.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> 
```

위의 이미지와 같이 sqlite3 test.db 를 입력하시면 해당 데이터베이스로 접속합니다. 현재 프로그램의 데이터베이스에서 가지고 있는 테이블은 아래를 참고하세요.

Meta_Data Table

<u>_id</u>	<u>_name</u>	<u>Ipfs hash</u>	<u>_date</u>	<u>_loca</u>	<u>Enc AES</u>	<u>Status</u>	<u>MDR id</u>

- ⇒ 각 속성(애트리뷰트)에 대해 설명을 드리겠습니다.
- ⇒ _id : insert 되는 순서대로 증가하는 autoincrement index
- ⇒ _name: clip 의 이름
- ⇒ Ipfs_hash : 암호화된 clip 을 ipfs add 했을 때에 반환되는 hash
- ⇒ _loca : ipfs camera node 의 위치
- ⇒ Enc_AES : clip 암호화에 사용된 AES_KEY 를 public_key 로 암호화한 결과
- ⇒ Status : 현재 메타데이터의 상태(0 : 생성만 됐을 시, 1: ipfs add, 암호화, merkleDir 갱신까지 모두 완료됐을 시)
- ⇒ MDR_id : merkleDir 의 primary key 를 참조하는 F.K(rootDir 의 id)

merkleDir Table

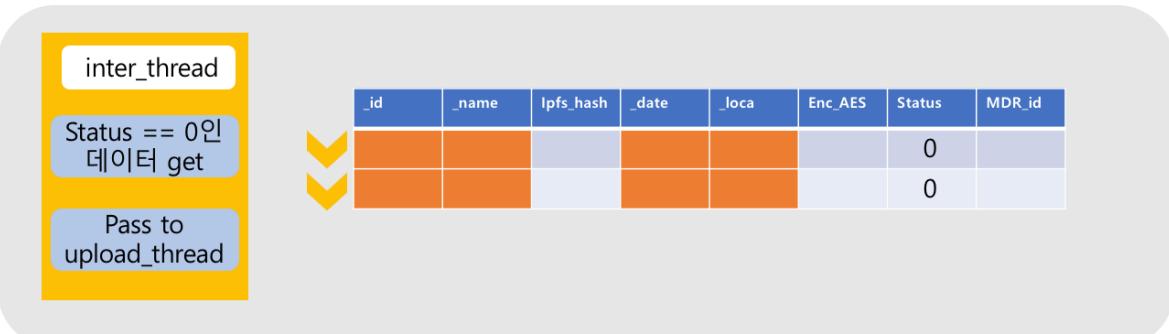
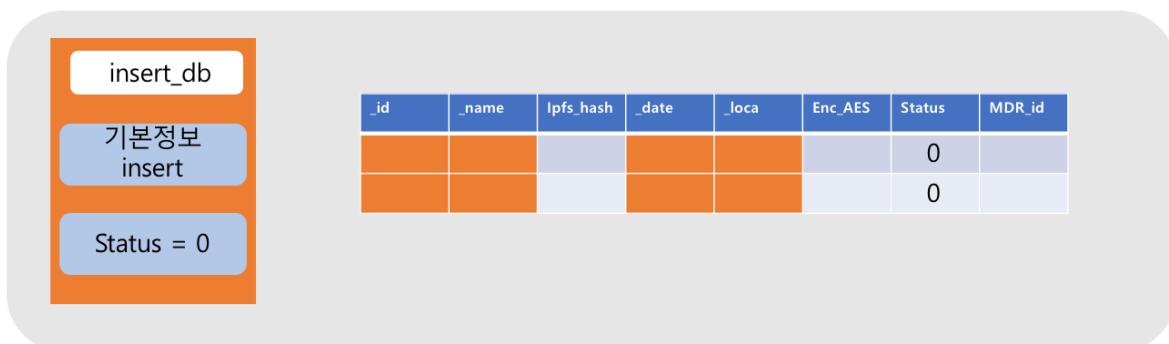
_id	Path	Hash
1	rootDir	...
2	rootDir/2018	...
3	rootDir/2018/02	...
4	rootDir/2018/02/20	...
5	rootDir/2018/02/21	...

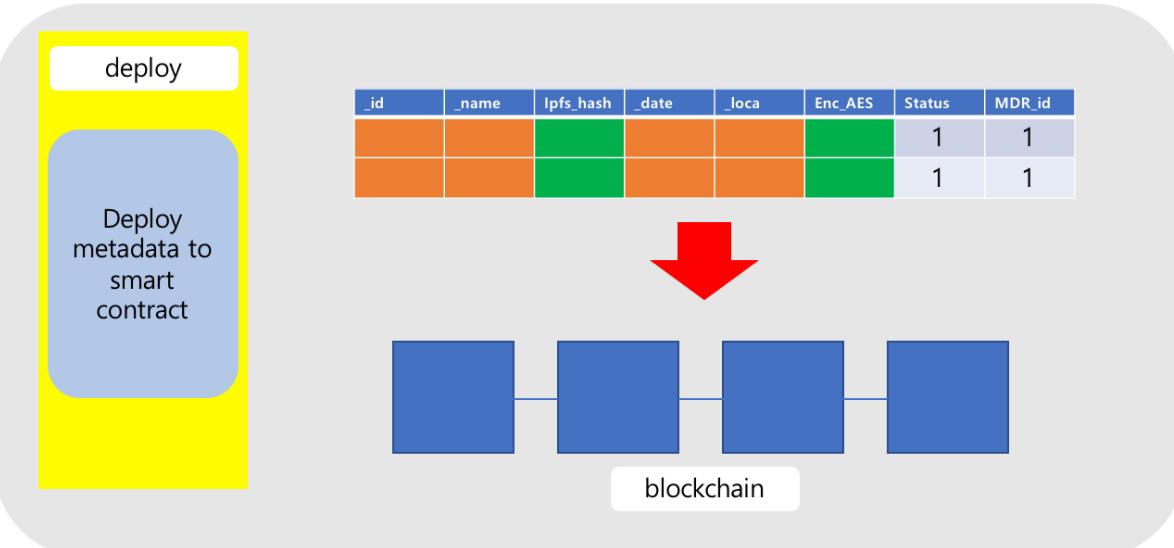
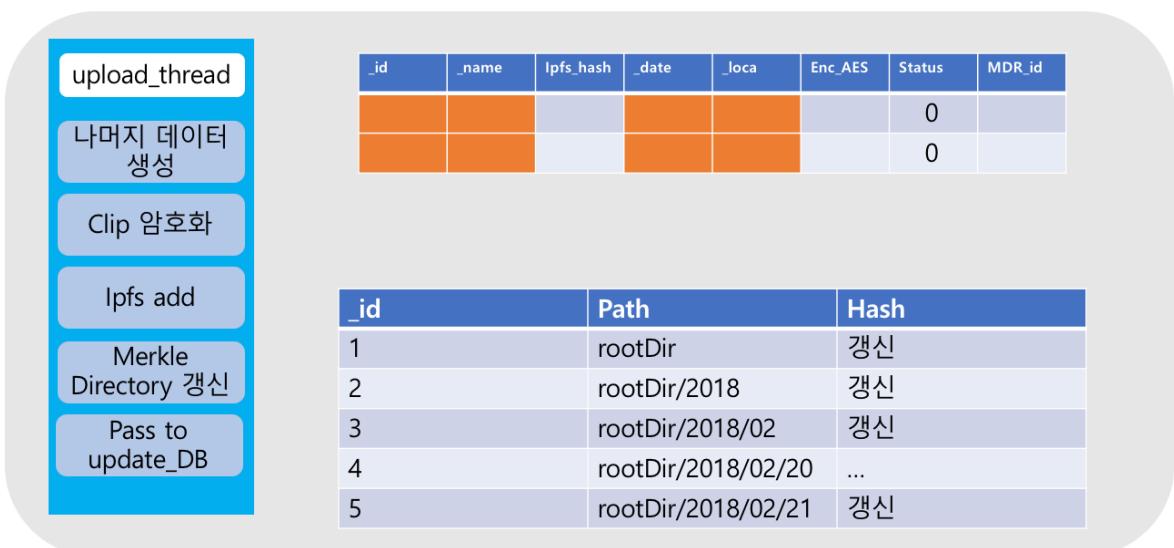
- ⇒ **merkleDir** 테이블입니다. 각 튜플은 예시로 적어두었습니다.
- ⇒ **_id** : insert 순서대로 증가하는 index(rootDir의 경우 항상 1)
- ⇒ **Path** : merkleDir의 경로
- ⇒ **Hash** : merkleDir의 hash (clip이 새로 생성될 때마다 갱신됨)

-Code

code 에는 각 주석을 상세하게 적어놓았습니다. 여기서는 모든 code block 들을 하나하나 자세하게 다루기 보다는 주요 기능에 대해서 확인하는 방향으로 작성하였습니다.

코드를 설명하기에 앞서 코드에 전체적인 흐름을 살펴보겠습니다.



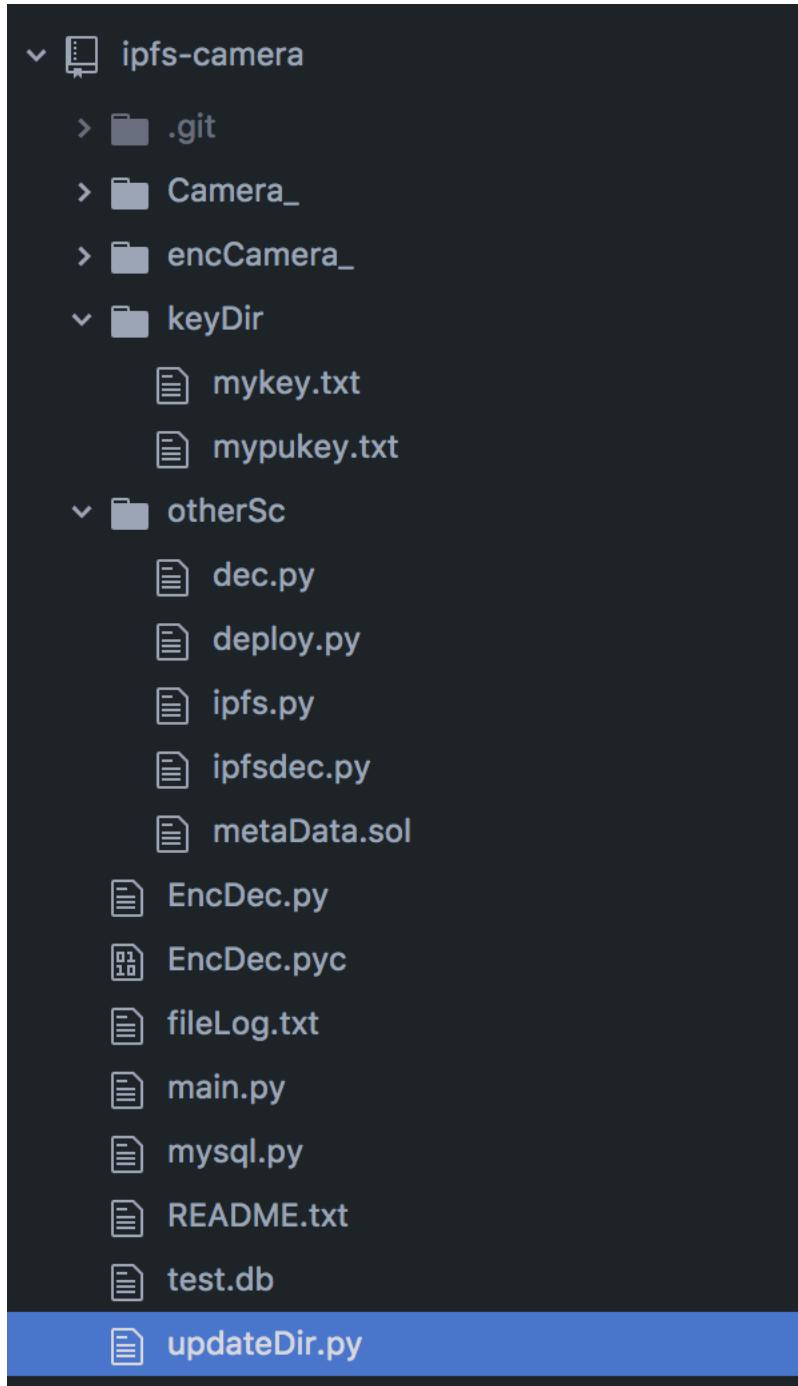


위의 나열된 이미지들이 전체적인 흐름입니다. 각각 다른 기능을 수행하는 6 개의 스레드 및 메소드가 있고 각자의 역할을 끝낸 뒤에 queue 를 이용해서 다음 스레드에 이후 작업을 넘겨줍니다. 각 작업에 따라 Meta_Data 테이블이 어떻게 갱신되는지 확인하실 수 있습니다.

1. 맨 처음 clip 이 생성되면 insert_db 스레드를 통해서 기본적인 정보(clip 의 이름, 생성날짜, 위치)를 삽입한 뒤에 이후 update 작업이 필요하다는 의미로 status 를 0 으로 초기화해줍니다.
2. Inter_thread 는 Meta_Data 에 삽입된 메타데이터들 중에 status == 0 인 데이터들을 update 를 위해 upload_thread 에 전달해주는 역할을 합니다. 만일 프로그램 수행중에 ipfs 가 비정상적으로 종료되어서 update 를 하지 못하는 경우, 프로그램 재실행시 먼저 Meta_Data 테이블을 살펴서 status == 0 인 데이터들(ipfs 비정상 종료시에 생성만 되고 update 를 하지 못한 clip 들)을 우선적으로 upload_thread 에 넘겨준 뒤에 새로 생성된 clip 의 데이터를 upload_thread 에 넘겨줍니다. 즉, 프로그램 재실행시에 생성되는 clip 들 보다 재실행 전에 생성되기만 하고 update 가 안된 데이터들을 우선처리 합니다.
3. Upload_thread 는 생성된 clip 들에 대해서 주요 작업을 처리하는 스레드입니다. 암호화를 수행하고 ipfs add 및 merkle directory 갱신 작업이 해당 스레드에서 수행됩니다. (merkle directory 갱신 작업은 엄밀히 말하면 updateDir.py 스크립트가 수행하지만 call 을 하는 주체가 upload_thread 입니다.)
4. Upload_thread 가 생성한 데이터들 (ipfs_hash, Enc_AES, merkleDir update)에 대해서 update_db 메소드가 데이터베이스를 갱신하는 작업을 수행합니다. 여기서 모든 필요한 데이터가 갱신되었음을 표시하는 status == 1 이 갱신됩니다.
5. Update_db 메소드가 메타데이터의 모든 칼럼값을 갱신한 뒤에 해당 메타데이터를 string화 하여 deploy 스레드에 넘겨줍니다.
6. Deploy 스레드는 전달받은 metadata 를 smart contract 에 전달하고 최종적으로 1 cycle 이 완료됩니다.

-기능별 script

먼저 프로젝트 디렉토리 내에 script 구성부터 설명드리겠습니다.



ipfs_camera라는 프로젝트 명 아래에 총 4개의 디렉토리(Camera_, encCamera_, keyDir, otherSc)가 있습니다.

- **Camera_** : 영상 clip 들이 저장되는 디렉토리입니다. 필요에 따라 위치를 외부 디렉토리로 바꿀 수 있습니다.
- **encCamera_** : Camera_ 내의 clip 들이 암호화 되어 저장되는 위치입니다. 마찬가지로 필요에 따라 위치를 외부 디렉토리로 변경 가능합니다.
- **otherSc** : 당장 프로그램을 실행하는데에 관련이 없는 script 들을 저장했습니다. Dec.py 의 경우 암호화되어 저장된 clip 들을 복호화 하는 test 를 수행하였고 ipfsdec.py 역시 마찬가지 입니다. 이후에 복호화 하는데에 혹시라도 필요할까봐 따로 저장해두었습니다. 또한 deploy.py 는 smart contract 를 새로 작성시에 blockchain 에 deploy 하는 코드입니다. Smart contract 코드의 테스트는 web complier 를 사용하시거나 따로 truffle framework 를 사용하시기를 권고하지만 단순히 transaction id, abi 를 추출하기 위해서는 위의 스크립트만 실행시키셔도 됩니다. Console 창에 transction id 와 abi 를 출력하도록 구성했습니다. 복사/붙여넣기로 활용하시면 됩니다. 그리고 metadata.sol 은 현재 프로그램에 사용된 smart contract 코드 입니다.
- **keyDir** : 암호화에 필요한 private key, public key 가 텍스트 형식으로 저장되어있습니다. Aes key 의 경우 영상 clip 하나당 생성되는데다 private key 로 enc_AES key 를 복호화 가능하기 때문에 따로 저장하지 않습니다.
- 그 외의 EncDec.py 부터 updateDir.py 까지는 프로그램을 실행하기 위한 스크립트입니다.

그 다음으로 프로그램을 실행하기위한 script 에 대해 정리하겠습니다.

- **Main.py** : 이름 그대로 프로그램을 실행하기 위한 메인 스크립트 입니다. 해당 스크립트에서 여러 기능들을 스레드화 하여 구동합니다.
- **EncDec.py** : main 스크립트가 실행하는데에 암호화 과정에서 호출하는 스크립트 입니다. 즉 library 로써 main.py 에 import 됩니다.
- **updateDir.py** : main 스크립트가 실행하는데에 머클 디렉토리화 수행 과정에서 호출하는 스크립트입니다. EncDec.py 와 마찬가지로 library 로써 main.py 에 import 됩니다.
- **mysql.py** : 현재 main 프로그램과는 상관이 없습니다. 다만 Mac mini 에서 smart contract 에 저장된 메타데이터를 mysql database table 에 저장하는 코드입니다. 독립적으로써 Mac mini 에서만 실행합니다.

- **test.db** : sqlite 의 데이터베이스 파일입니다. 1 분간격 영상의 메타데이터를 저장하기 위한 metaData 테이블, 머클 디렉토리 데이터를 저장하는 Camera1, Camera2, Camera3(카메라 이름으로써 임의로 설정했습니다)테이블이 존재합니다.
- **Filelog.txt** : 일전에 모니터링 프로그램을 만들 때에 영상이 제대로 저장되는지 log 를 남기기 위한 텍스트 파일입니다. 현재는 database table 에 저장하기 때문에 큰 필요성은 없습니다.
-

-기능별 code

코드를 살펴보겠습니다.

```

Cameralpath = '/Users/leebongho/monitoring/Camera_/'          #Car
fileLog='/Users/leebongho/monitoring/fileLog.txt'            #fi
encDir = '/Users/leebongho/monitoring/encCamera_/'           #enc
metaData = '/Users/leebongho/monitoring/metaData.txt'         #meta
conn = sqlite3.connect('test.db', check_same_thread=False)      #Con
cur = conn.cursor()                                           #Dat
insertQ = Queue()                                            #Que
deployQ = Queue()                                           #Upd
daemonQueue = Queue()                                         #exec
now = datetime.now()                                         #sta
uploadQ = Queue()                                           #to
waitQ1 = Queue()                                            #wa
waitQ2 = Queue()                                           #wa

```

위의 코드는 프로그램 실행시에 영상파일 및 암호화 영상파일 저장 위치를 지정하고 sqlite DataBase 를 연결한 뒤 cursor 객체를 지정하는 코드입니다.

그리고 thread 간 통신을 위한 queue 를 총 6 개 정의합니다. 이 중에 waitQ 에 해당하는 queue 는 스레드간 데이터베이스의 공동참조를 방지하기위한 대기 큐입니다.

insertQ : 모니터링 스레드가 캐치한 영상의 이름을 저장합니다.

deployQ : 영상의 메타데이터를 smart contract 에 저장하기 위해 deploy 스레드에 전송합니다.

daemonQueue : 프로그램 실행중 IPFS repo lock 에러 발생시 lock file 을 제거 후 ipfs daemon 을 재 실행시켜주는 daemon 스레드 실행 트리거 입니다.

uploadQ : status == 0 인 메타데이터의 이름을 전달받아서 이후 작업처리를 위해 upload_thread 에 전달하는 queue 입니다.

waitQ1 : insert_db 스레드가 기본데이터 insert 를 하기 전에 inter_thread 가 이전에 생성만 되고 update 를 하지 못한 데이터들을 우선적으로 검색하기 위해 insert_db 가 대기하게끔 합니다.

waitQ2 : insert_db 가 새로 생성된 clip 들에 대해 기본정보를 insert 하고 그 사이에 inter_thread 가 데이터베이스를 참조하지 못하게 대기시킵니다.

```
rpc_url = "http://192.168.1.2:8545" #Mac mini의 geth와 통신하기 위한 HTTPProvider
w3 = Web3(HTTPProvider(rpc_url, request_kwargs={'timeout': 500}))
contract = w3.eth.contract(abi=[{'constant': True, 'inputs': [{"name': 'id', 'type': 'uint256"}], 'name': 'get', 'type': 'uint256'}], 'payable': False, 'stateMutability': 'view', 'type': 'function'})
```

rpc_url 은 geth 동기화를 유지하는 BlockChain node(Mac mini)의 주소입니다. mac mini 의 chain-data 를 쓰기 위해서 HTTPProvider 를 명시해줍니다.

w3 은 web3 객체로써 사용합니다. timeout 인자는 blockchain 과의 통신을 500 초까지 대기합니다.

contract 객체 생성을 해줍니다. abi 의 경우 smart contract 를 compile 시에 반환하는 값으로써, 해당 abi 가 없으면 contract 와의 통신이 불가능 합니다. (아니면 매 프로그램 실행 시 마다 solidity code 를 컴파일 해주어야 합니다.)

```

def Camera(i) :
    os.chdir(Camerapath)      #Camera_ 디렉토리에서 해당 프로그램 실행을 위한 경로 설정
    i=i+1                     #프로세스가 원치않게 종료되었을 때 영상 파일 이름을 명시하기 위한 변수
    try:
        sub = subprocess.check_output('openRTSP -D 3 -B 250000 -b 250000 -c -i -F aCAM'+ str(i) +' -P 90 -u admin admin rtsp://192.168.1.26/11', stderr=subprocess.STDOUT, shell=True)
    except:
        print("error")
    Camera(i)                 #원치않게 스레드가 종료되었을 때 다시 실행하기 위해서 재구호출

```

이전에 정의해둔 Camerapath 를 프로그램 실행 디렉토리로 설정합니다.
 openRTSP 를 실행함으로써 IP Camera 의 영상 clip 을 해당 디렉토리에 저장합니다.
 예기치 않게 openRTSP 가 종료되는 경우에 예외처리를 통해서 재실행 하게끔
 구현했습니다.

```

class LogHandler(PatternMatchingEventHandler) :          #모니터링 프로그램의 클래스
    def __init__(self) :                                #생성자 호출
        super(LogHandler, self).__init__(ignore_patterns=["*/ex.log"])
        f = open(fileLog, 'w')                          #해당 스레드가 실행될 때마다 fileLog의 내용을 초기화 시켜주기 위함
        f.seek(0)
        f.truncate()
        f.close()
    def on_created(self, event) :                      #created 이벤트 발생시에 수행할 작업
        super(LogHandler, self).on_created(event)
        what = 'directory'
        self.eventLog = "created, " + event.src_path
        print(self.eventLog)
        with open(fileLog, 'a') as fout:                #생성된 영상파일의 로그를 텍스트 파일에 저장(큰 필요 없음)
            fout.write(self.eventLog.split('/')[-1])
            fout.write('\n')
            fout.close()
        queue.put(self.eventLog.split('/')[-1])         #중요함. 캐치한 이벤트 (생성된 파일)의 이름을 queue에 삽입

```

monitoring 스레드를 실행하기 위한 thread 입니다. monitoring 스레드의 경우
 파일의 외부모듈 watchdog 을 통해 수행이 됩니다. on_created 메소드는
 디렉토리에 created 이벤트 발생시 해당 파일의 이름을 queue 에 삽입해줍니다.

```
def inter_thread() :
    temp = ""
    sql = 'SELECT _name FROM Meta_Data WHERE status=0'
    cur.execute(sql)
    rows = cur.fetchall()
    for row in rows :
        clip_name = row[0]
        uploadQ.put(clip_name)
        temp = clip_name
    waitQ1.put(1)
    while True :
        wait = waitQ2.get() #wait for insert_
        sql = 'SELECT _name FROM Meta_Data ORDER BY _id DESC LIMIT 1;'
        cur.execute(sql)
        fetch_name = cur.fetchone() #select a
        name = fetch_name[0] #convert
        if temp == name : #if newly made
            continue
        else : #if newly
            uploadQ.put(name) #put to up
            temp = name #temp keep
            continue
```

Inter_thread 입니다. 역할은 위해서도 말했듯이 Meta_Data 테이블을 검색하여 status == 0 인 데이터들을 갱신하기 위해 먼저 upload_thread 에 전달하는 역할을 합니다. Temp="" ~~ while 까지를 먼저 살펴보겠습니다. 해당 코드가 clip 이 생성만 되고 update 가 되지 않은 상태의 status == 0 인 메타데이터들을 우선적으로 검색해서 uploadQ 에 삽입해주는 과정입니다. 해당 작업을 수행한 뒤에 waitQ1 을 풀어줌으로써 접근 권한이 insert_db 에 넘어갑니다. 해당 코드는 프로그램을 실행했을 때 딱 1 번 실행이 됩니다.

```
def insert_db() :
    wait = waitQ1.get()                                #wait until inter_thread put all Data.
    while True :
        name = insertQ.get()                          #start insert to DB table newly
        print('basic insert start')
        ctime = os.path.getctime(Camerapath + name)    #To insert create date of clip
        dt = datetime.fromtimestamp(ctime)              #To insert create date of clip
        g = geocoder.ip('me')                          #To insert location of clip
        date = str(dt)                               #transform to string type
        loca = str(g.latlng)                         #transform to string type
        status_flag = 0                               #set to status (status=0 means
        query = 'INSERT INTO Meta_Data(_name, _date, _loca, status) VALUES(?, ?, ?, ?)'
        cur.execute(query, (name, date, loca, status_flag))
        conn.commit()
        waitQ2.put(1)                                #wait2 Queue unlock to newly created
```

Meta_Data 테이블의 접근 권한이 insert_db 스레드에 넘어왔으니 해당 코드를 살펴보겠습니다. While 문 내에서 insert.get()을 통해 monitoring 스레드가 넘겨준 생성된 clip 들의 이름을 가져옵니다. 이후 기본적인 데이터(_name, _date, _loca)를 삽입하기 위해 파일의 생성시간 및 위치를 각각 date, loca 에 저장합니다. 그리고 현재 기본적인 데이터만 삽입되었다는 의미를 알려주는 status_flag 를 0 으로 초기화합니다. 해당 과정을 마치고 나면 Meta_Data 에 기본 데이터가 삽입이 되고 waitQ2 의 대기를 풀어줌으로써 inter_thread 가 그 다음작업을 수행할 수 있게됩니다.

```
def inter_thread() : #inter_th
    temp = "" #To store
    sql = 'SELECT _name FROM Meta_Data WHERE status=0' #SELECT a
    cur.execute(sql)
    rows = cur.fetchall()
    for row in rows : #store to
        clip_name = row[0] #store cl
        uploadQ.put(clip_name) #put clip
        temp = clip_name #latest c
    waitQ1.put(1) # waitQ1 Queue u
    while True : #waiti for insert_
        wait = waitQ2.get() #wati for insert_
        sql = 'SELECT _name FROM Meta_Data ORDER BY _id DESC LIMIT 1;' #select a
        cur.execute(sql) #select a
        fetch_name = cur.fetchone() #convert
        name = fetch_name[0] #convert
        if temp == name : #if newly me
            continue #if newly me
        else : #if newly
            uploadQ.put(name) #put to u
            temp = name #temp kee
            continue
```

이번에 살펴볼 내용은 접근 권한을 허락받은 inter_thread 의 이후 작업입니다. 박스 내의 코드들이 실행이 됩니다. Insert_db 가 새로 생성된 데이터들의 기본 정보를 Meta_Data 테이블에 삽입을 해주었으니(status == 0) 이제 기본 정보가 입력된 데이터들을 갱신하는 작업을 처리해야 합니다. Meta_Data 테이블로부터 최신 데이터(방금 insert_db 가 삽입한 데이터)를 가져와서 uploadQ 에 삽입하여줍니다. 가져온 데이터가 최신데이터가 맞는지에 대해서 확인하기 위해 temp 를 이용해 비교합니다.

```

def upload_thread(temp_year, temp_month, temp_day) :
    time.sleep(10)                                     #An upload thread that handles the main task, encrypts the clip, receives the h

    while True:                                         #loop
        now = datetime.now()                           #이후 카메라별 IPFS 디렉토리를 갱신할 때에 월/별 구분하기 위해 현재날짜 객체를 생성
        check = uploadQ.queue[0]                      #get the value stored in uploadQ (not queue.get(), not pop from queue)
        temp = os.path.getsize(Camerapath + check)    #store current size of clip to check if streaming is complete
        time.sleep(5)                                    #wait for 5 seconds
        checkSize = Camerapath + check                #store size of clip after 5 seconds
        if os.path.getsize(checkSize) == temp :         #compare sizes changes between 5 seconds (if not equal, streaming is not
            toAdd = uploadQ.get()                     #get complete(streaming) clip from uploadQ.
            print(toAdd)
            AES_key = EncDec.Random.new().read(32)      #create AES_KEY to encryption(clip encryption)
            enc_key = EncDec.rsa_enc(AES_key)           #encrypt AES_KEY using public_key (enc_key == encrypted AES_KEY : Enc
            dec_key = EncDec.rsa_dec(enc_key)           #to decrypt afterwards, (dec_key == AES_KEY)
            in_filename = Camerapath + toAdd.strip()    #store the name of the clip to be encrypted(즉 Camera_/영상파일)
            os.chdir(encDir)                          #work directory is encCamera_(encrypted clip will be created in this d
            EncDec.encrypt_file(AES_key, in_filename, out_filename=toAdd.strip())  #encrypt the clip using AES_KEY
            print('enc!!!!')
            print('srtat IPFS Add')
            time.sleep(2)                                #wait for 2 seconds
            try :
                ipfsAdd=subprocess.check_output('/usr/local/bin/ipfs add ' + encDir + toAdd.strip(), stderr=subprocess.STDOUT)
            except :
                daemonQueue.put(1)                         #if ipfs daemon was terminated, restart
                time.sleep(30)
                ipfsAdd=subprocess.check_output('/usr/local/bin/ipfs add ' + encDir + toAdd.strip(), stderr=subprocess.STDOUT)
            clip_name=ipfsAdd.decode().split(' ')[-1].strip()          #store encrypted clip_name(equal to original
            time.sleep(30)
            print('IPFS Add Done. updating Merkle Directory')          #ipfs add done, start merkle directroy update

```

주요 작업을 처리하는 upload_thread 입니다 inter_thread 가 넘겨준 데이터에 대해서 암호화 및 ipfs add, merkle directory 갱신 작업을 수행합니다. 먼저 clip 이 streaming 이 완료가 되었는지 확인을 하기 위해서 용량을 비교합니다. 5 초 사이에 용량 변화가 있었다면 아직 streaming 중이라고 판단하여 다시 시도합니다. 만약 5초사이에 용량 변화가 없다면 streamin 이 완료되었다고 판단하여 if 문 내의 코드를 실행합니다.

toAdd 라는 변수에 uploadQ 로부터 가져온 갱신할 clip 의 이름을 가져옵니다. 그리고 암호화를 위해 AES_key 를 생성해줍니다. 이제 Meta_Data 테이블을 업데이트 하기 위한 Enc_AES 가 마련되었습니다. 그 다음 필요한 내용은 암호화된 clip 의 ipfs_hash 와 merkle directory 의 갱신입니다.

암호화된 clip 이 생성 되었으니 단순히 ipfs add 를 해주면 hash 값을 얻을 수 있습니다. 여기서 예기치않게 ipfs 가 종료되는 버그 및 오류가 발생했을 시에 try, except 를 이용한 예외처리로 daemon 스레드를 다시 구동한 뒤 ipfs add 작업을 수행합니다. 여기까지 완료되었다면 Enc_AES 와 ipfs_hash 필드를 업데이트하기 위한 값을 얻었습니다. 이제 merkleDir 만 업데이트 하면 됩니다. 잠깐 그 전에 clip 암호화와 관련된 EncDec.py 스크립트의 코드를 짚고 넘어가겠습니다.

```
#AES KEY를 이용해서 영상을 암호화 하기 위한, key는 AES_KEY, in_filename은 영상 이름 삽입
def encrypt_file(key, in_filename, out_filename=None, chunksize=65536):
    if not out_filename:
        out_filename = in_filename + '.enc'
    iv = b'initialvector123'           #초기화 벡터를 따로 지정해줌, 이후 복호화 할 때에도 무조건 필요
    encryptor = AES.new(key, AES.MODE_CBC, iv) #AES key와 초기화벡터(iv)를 이용해서 CBC모드로 암호화 하기 위한 객체
    filesize = os.path.getsize(in_filename)      #암호화 하려는 영상 파일의 크기를 가져옴
    with open(in_filename, 'rb') as infile:       #영상 파일의 내용을 infile이라는 이름으로 저장
        with open(out_filename, 'wb') as outfile:   #원본 영상 파일을 암호화파일로 만들기 위한
            outfile.write(struct.pack('<Q', filesize)) #struct.pack은 형변환. 즉 원본파일의 사이즈 만큼 형변환해서 암호화
            outfile.write(iv)                      #초기화 벡터를 같이 넣어준다.
            while True:
                chunk = infile.read(chunksize)      #원본파일을 chunksize만큼 읽어서 chunk변수에 저장
                if len(chunk) == 0:                  #chunk 사이즈가 0이라면 (읽어온 내용이 없다면) 암호화 종료
                    break
                elif len(chunk) % 16 != 0:           #읽어온 사이즈가 16바이트로 나누어 떨어지지 않는다면 나누어 떨어질만큼 공백을 삽입
                    chunk += b' ' * (16 - len(chunk) % 16)
                outfile.write(encryptor.encrypt(chunk)) #chunk크기만큼 AES_key로 암호화
```

encryption 을 위한 외부모듈로써 **pycrypto** 를 사용했습니다. 암호화 관련된 스크립트 코드는 EncDec.py 로 따로 모듈화를 했습니다.

위의 코드는 EncDec.py 에서 파일 암호화(영상 clip)에 해당하는 encrypt_file 메소드입니다. 이전에 생성된 **AES_KEY**, **암호화 하기 위한 clip**, **암호화 된 clip** 의 이름을 인자로 넘겨줍니다. chunksize 는 원본 clip 에서 해당 크기 만큼 암호화를 진행하기 위해 지정합니다.

iv 의 경우 초기화 벡터로써, 이후 복호화에서도 반드시 필요합니다. encryptor 객체로써 암호화에 사용하기 위해 AES_KEY 와 iv 를 인자로 넘겨줍니다.

이후 영상 clip 파일을 읽어서 암호화를 수행한 뒤 이전에 인자로 넘겨준 이름으로 암호화된 파일을 만들어줍니다..

암호화 과정을 마치고 해당 머클 디렉토리를 구성하기 위한 과정입니다

merkleDir 을 구성하는 코드는 양이 많기 때문에 나누어서 설명드리겠습니다..

```

ctime = os.path.getctime(Camerapath + clip_name)           #to store creation time of the clip
temp = str(datetime.fromtimestamp(ctime))                  #convert to string
temp1 = temp.split(' ')[0].replace("-", "/")               #convert to year/month/day
day_dir = temp1.split('/')[2]                             #day_dir store day
month_dir = temp1.split('/')[1]                           #month_dir store month
year_dir = temp1.split('/')[0]                            #year_dir store year

day_path = 'rootDir'+'/' +year_dir+ '/' +month_dir+ '/' +day_dir      #day_path is rootDir/year/month/day
month_path = 'rootDir'+'/' +year_dir+ '/' +month_dir                #month_path is rootDir/year/month
year_path = 'rootDir'+'/' +year_dir                                #year_path is rootDir/year
sql = 'SELECT EXISTS (SELECT * FROM merkleDir WHERE path=?)'      #make sure it already exists
day = (day_path,)
month = (month_path,)
year = (year_path,)
cur.execute(sql,day)                                              #Verify that the day directory exists
check_day = cur.fetchone()[0]
cur.execute(sql,month)                                             #Verify that the month directory exists
check_month = cur.fetchone()[0]
cur.execute(sql,year)                                              #Verify that the year directory exists
check_year = cur.fetchone()[0]

```

먼저 clip 이 생성된 시간 (year/month/day)으로 머클 디렉토리를 구성하기 때문에 day_dir, month_dir, year_dir 이름의 변수에 각각 파일 생성된 시간을 넣어줍니다. 예를들어 2018-02-11 생성된 파일이라면 day_dir = 11, month_dir = 02, year_dir = 2018 이 되겠습니다.

그리고 merkleDir 을 간신하기 위해 각 directory 의 경로를 설정합니다 rootDir/year, rootDir/year/month,rootDir/ year/month/day 가 되겠습니다.

그리고 sql 쿼리를 이용해서 현재 merkleDir 테이블에 해당 경로가 존재하는지를 체크합니다. 즉, 파일 생성 시점이 11 일에서 12 일로 넘어간 첫 시점이라면 merkleDir 테이블에는 2018/02/11 까지만 있고 2018/02/12 는 새로 만들어주어야 합니다. 파일 생성 날짜와 일치하는 path 가 존재한다면 단순히 삽입(link)시켜주면 됩니다.

Check_day, check_month, check_year 는 각 디렉토리가 존재하는지의 여부를 반환합니다. 존재한다면 1 을, 존재하지 않는다면 0 을 반환합니다.

이제 존재 여부에 따라 다른 함수를 호출합니다. 그 과정을 살펴보겠습니다.

```
if check_year == 1 :      #if year directory already exists in merkleDir table, check month
    if check_month == 1 :   #if month directory already exists in merkleDir tble, check day
        if check_day == 1 :  #if year/month/day directory already exists in merkleDir
            root_hash = updateDir.dirUpdate1(year_dir, month_dir, day_dir, ipfsAdd)
        elif check_day == 0 : #if year/month directory already exsits but year/month/day directory does not exist
            root_hash = updateDir.dirUpdate1_1(year_dir, month_dir, day_dir, ipfsAdd)
        elif check_month == 0 : #if year directory already exsits but year/month directory does not exist
            root_hash = updateDir.dirUpdate2(year_dir, month_dir, day_dir, ipfsAdd) #
        elif check_year == 0 : #if year directory, year/month directory and year/month/day directory does not exist
            root_hash = updateDir.dirUpdate3(year_dir, month_dir, day_dir, ipfsAdd)
    print('Update merkle Direcotry Done. start deploying to Smart contract')           #merkleDir was updated
    update_db(clip_name, ipfsAdd, enc_key)                                              #pass the arguments to update_db
    uploadQ.task_done()                                                                #uploadQ was done
    print('DB update task done')
    os.remove(Camerapath + toAdd)                                                       #if update Database, remove updated clip from camera
    os.remove(encDir + toAdd)                                                          #if update Database, remove updated clip from encDir
```

위의 코드에서 바로 이어지는 코드입니다. 조건문에 따라 check_year부터 검사합니다. 현재 merkleDir에 year에 해당하는 디렉토리 (rootDir/2018)이 존재한다면 month 디렉토리 (rootDir/2018/02)까지 존재하는지, 역시 존재한다면 day 디렉토리 (rootDir/2018/02/12)가 존재하는지 차례로 검사합니다.

모두 참이라서 rootDir/2018/02/12가 존재한다면 단순히 clip을 day부터 차례로 link 시켜주기 위해 updateDir.py의 dirUpdate1 함수를 call 합니다.

만일 rootDir/2018/02까지는 참이지만 rootDir/2018/02/12, 즉 day에 해당하는 디렉토리가 없다면 updateDir.py의 dirUpdate1_1 함수를 call 합니다.

차례로 day가 없을때, month가 없을때, year가 없을 때의 call 하는 함수들은 updateDir.py 스크립트에 있습니다. 확인하겠습니다.

```

def dirUpdate1(temp_year, temp_month, temp_day, ipfsAd) :           #When year / month / day already exists
    clip_name=ipfsAd.decode().split(' ')[-1].strip()      #Extract clip name
    clip_hash=ipfsAd.decode().split(' ')[-2]                #Extract clip hash
    sql="SELECT hash FROM merkleDir WHERE path=? "        #Get the ipfs hash corresponding to path in the merkleDir ta
    day_path = ('rootDir/'+str(temp_year)+'/'+str(temp_month)+'/'+str(temp_day),)   #store year/month/day path to
    cur.execute(sql, day_path)                            #run sql
    day_hash = str(cur.fetchone()[0])       #Save the year/month/day hash returned by sql as day_hash.
    month_path=( 'rootDir'+'/'+str(temp_year)+'/'+str(temp_month),) #store year/month path to select matched hash from
    cur.execute(sql, month_path) #run sql
    month_hash = str(cur.fetchone()[0]) #Save the year/month hash returned by sql as month_hash
    year_path=( 'rootDir'+'/'+str(temp_year),) #store year path to select matched hash from merkleDir
    cur.execute(sql, year_path)#run sql
    year_hash = str(cur.fetchone()[0]) #Save the year hash returned by sql as year_hash
    root_path = ('rootDir',)           #same about rootDir
    cur.execute(sql, root_path)
    root_hash = str(cur.fetchone()[0])
    up_d = subprocess.check_output('/Users/leebongho/work/bin/ipfs object patch '+ day_hash +' add-link ' + clip_n
    sql = 'UPDATE merkleDir SET hash=? WHERE path=? '          #Query to update the modified hash(day).
    update=(str(up_d), 'rootDir/'+str(temp_year)+'/'+str(temp_month)+'/'+str(temp_day)) #table update about modifi
    cur.execute(sql, update)
    conn.commit()
    up_m = subprocess.check_output('/Users/leebongho/work/bin/ipfs object patch '+ month_hash +' add-link ' + str(t
    update=(str(up_m), 'rootDir/'+str(temp_year)+'/'+str(temp_month))   #Query to update modified hash(month/day)
    cur.execute(sql, update)
    conn.commit()      #업데이트 사항 저장
    up_y = subprocess.check_output('/Users/leebongho/work/bin/ipfs object patch '+ year_hash +' add-link ' + str(t
    update=(str(up_y), 'rootDir'+'/'+str(temp_year))             #Query to update modified hash(year/month)
    cur.execute(sql, update)
    conn.commit()
    up_r = subprocess.check_output('/Users/leebongho/work/bin/ipfs object patch '+ root_hash +' add-link ' + str(t
    update=(str(up_r), 'rootDir')                                #Query to update modified hash(rootDir/year/month)
    cur.execute(sql, update)
    conn.commit()

```

먼저 dirUpdate1 함수입니다. 코드가 길어보이지만 중복된 내용이 많아서 실제로는 많지 않습니다. dirUpdate1 함수는 year/month/day 가 모두 존재할 때 호출됩니다. 따라서 생성된 clip 을 단순히 year/month/day 밑에 link 를 시켜주고 상위 디렉토리들의 hash를 갱신합니다.

emptyDir 의 경우 IPFS 빈 디렉토리의 hash 입니다. 머클 디렉토리에 새로운 year, month 디렉토리를 link 시에 해당 hash를 사용합니다.

우선 IPFS object path 를 통해 머클 디렉토리를 갱신하기 위해서는

- 1) link 하려는 영상 clip 의 이름과 해시, clip 의 부모 노드인 day 의 해시가 필요합니다. 그리고
- 2) 영상 clip 이 저장됨에 따라 갱신된 day 의 이름과 해시, day 의 부모 노드인 month 의 해시가 필요합니다.
- 3) 이전과 마찬가지로 갱신된 day 가 link 됨에 따라 갱신된 month 디렉토리의 hash 와 이름, 부모 노드인 year 디렉토리의 hash 가 필요합니다.
- 4) 같은 작업을 갱신된 month 디렉토리 및 기존 year 디렉토리에 대해 수행하여 줍니다. 갱신된 month 가 link 됨에 따라 갱신된 year 디렉토리의 hash 와 이름, 부모 노드인 root 디렉토리의 hash 가 필요합니다.
- 5) 최종적으로 갱신된 year 해시가 root 디렉토리에 link 됨으로써 갱신된 root 해시가 반환됩니다.

이에 따라 머클 디렉토리 테이블에 저장되는 내용은 root 디렉토리의 경로와 해시 및 각 year, month, day 의 경로와 해시입니다.

```
sqlite> select * from Camera1;  
rootDir|QmU5yeQ1iRTjooTfEdMe8FRZFA75hKzRV6sq4sdBnugLW1  
rootDir/2018|QmYrG5djujUgSvN9jFJUSYXmJD3ovtkLPbxC3VwNqeLSN  
rootDir/2018/1|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2|QmfNRoBccja4jseZ9NNpqPBfEEoAVKQTCrjRU3TUNaiUxV  
rootDir/2018/2/1|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/2|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/3|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/4|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/5|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/6|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/7|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/8|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/9|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/10|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/11|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/12|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn  
rootDir/2018/2/13|QmUNLLsPACCz1vLxQVkJXqqLX5R1X345qqfHbsf67hvA3Nn
```

위의 이미지가 현재 Camera1(임의로 설정)의 머클디렉토리 테이블입니다.

디렉토리의 경로 및 해시를 확인하실 수 있습니다.

즉,

1. Day 디렉토리에 영상 clip 을 link

필요한 것 : 기존 Day 디렉토리의 <hash>, 영상 clip 의 <hash, <name>

반환되는 것 : 갱신된 (영상 clip 이 link 된) Day 디렉토리의 <hash>

2. Month 디렉토리에 갱신된 day 디렉토리를 link

필요한 것 : 기존 month 디렉토리의 <hash>, 갱신된 day 의 <hash> <name>

반환되는 것 : 갱신된 (영상 clip 이 link 된) month 디렉토리의 <hash>

3. Year 디렉토리에 갱신된 month 디렉토리를 link

필요한 것 : 기존 year 디렉토리의 <hash> , 갱신된 month 의<hash> <name>

반환되는 것 : 갱신된 (month 디렉토리가 link 된) year 디렉토리의 <hash>

4. Root 디렉토리에 갱신된 year 디렉토리를 link

필요한 것 : 기존 root 디렉토리의 <hash>, 갱신된 year 의 <hash> <name>

반환되는 것 : 갱신된 (year 디렉토리가 link 된) root 디렉토리의 <hash>

```

if check_year == 1 :      #if year directory already exists in merkleDir table, check mo
    if check_month == 1 :   #if month directory already exists in merkleDir table, check
        if check_day == 1 :  #if year/month/day directory already exists in merkleDir
            root_hash = updateDir.dirUpdate1(year_dir, month_dir, day_dir, ipfsAdd)
        elif check_day == 0 : #if year/month directory already exists but year/month/day
            root_hash = updateDir.dirUpdate1_1(year_dir, month_dir, day_dir, ipfsAdd)
        elif check_month == 0 : #if year directory already exists but year/month directory
            root_hash = updateDir.dirUpdate2(year_dir, month_dir, day_dir, ipfsAdd) #
        elif check_year == 0 : #if year directory, year/month directory and year/month/day directory
            root_hash = updateDir.dirUpdate3(year_dir, month_dir, day_dir, ipfsAdd)
    print('Update merkle Directory Done. start deploying to Smart contract')               #me
    update_db(clip_name, ipfsAdd, enc_key)                                              #pass the arguments to update_db
    uploadQ.task_done()                                                                #uploadQ was done
    print('DB update task done')
    os.remove(Camerapath + toAdd)           #if update Database, remove updated clip from
    os.remove(encDir + toAdd)              #if update Database, remove updated clip from

```

해당 과정을 마치면 upload_thread 는 결과적으로 ipfs_hash, Enc_AES, MDR_id 를 갱신하기 위한 모든 데이터를 가지고 있습니다. 이제 이 데이터를 가지고 Meta_Data 테이블을 갱신하기 위해 update_db 메소드를 call 합니다. 그리고 이제 Meta_Data화 완료된 clip 들을 Camera_ 및 encCamera_ 디렉토리에서 삭제합니다.

Update_db 메소드를 살펴보겠습니다.

```
def update_db(clip_name, ipfsAdd, enc_key) : #update_db method : get status
    print('DB update start')
    sql = 'UPDATE Meta_Data SET ipfs_hash=?, Enc_AES=?, status=?, MDR_id=? WHERE _name=?'
    status_flag = 1
    merkle_root = 1
    clip_hash=ipfsAdd.decode().split(' ') [-2]
    upDB = (clip_hash, str(enc_key), status_flag, merkle_root, clip_name)
    cur.execute(sql, upDB)
    conn.commit()
    sql = 'SELECT * FROM Meta_Data WHERE _name=?'
    selDB = (clip_name,)
    cur.execute(sql, selDB)
    inqueue = cur.fetchone()
    deployQ.put(str(inqueue))
```

Clip 이름(clip_name), ipfs_hash(ipfsAdd), Enc_AES(enc_key)을 받아서 Meta_Data 테이블을 업데이트합니다. 또한 status 필드를 0 -> 1로 갱신합니다.

이후 갱신된 메타데이터를 string화하여 deploy 스레드에 인자로 넘겨줍니다.

```
def deploy() : #스마트컨트랙트에 메타데이터를 저장하기 위한 스레드
    while True :
        i = 0
        setData = queue2.get() #queue2에 저장된 데이터(데이터베이스에서 추출한 마지막 열의 메타데이터)를 setData에 저장
        tx_receipt = w3.eth.getTransactionReceipt('0xd501b20ee29b361f7babde65bbc78a13b583e8936a09aa235cde71289c39ebdb') #스마트 컨트랙트의 주소를 추출
        contract_address = tx_receipt['contractAddress']
        contract_instance = contract(contract_address)#컨트랙트 주소를 이용해서 컨트랙트 인스턴스 생성
        # Set
        tx = contract_instance.transact({"from": w3.eth.accounts[0],"gas": 500000}).insertData(str(setData))#스마트컨트랙트에 setData를 저장함. 저장하는 과정에서 트랜잭션이 체결될 때 까지 대기
        print('smart contract value inserted value : {}'.format(setData))
        while w3.eth.getTransactionReceipt(tx) is None : #트랜잭션이 체결될 때 까지 대기
            time.sleep(3)

        temp = contract_instance.call().getIndex() #컨트랙트내의 배열 인덱스를 가져옴, 이 인덱스를 가지고 컨트랙트 배열에 저장된 메타데이터를 추출
        print('last index : {}'.format(temp))
        print('inserted value get : {}'.format(contract_instance.call().getData(temp)))
        queue2.task_done()
        print('deploy task done')
```

메타데이터를 smart contract에 삽입하기 위한 deploy 스레드입니다. 위에서 queue에 삽입한 메타데이터를 get 합니다. 그리고 최초에 smart contract를 배포할 때에 반환받은 트랜잭션 id를 이용해서 배포한 contract의 주소를 얻으면 해당 contract와 통신할 수 있는 객체 생성이 가능합니다.

객체를 생성했으니 이제 메타데이터를 컨트랙트에 삽입하는 과정만 남았습니다. w3.eth.accounts[0]은 geth로 생성한 첫 번째 지갑의 주소로써, 이 계정을 이용해서 트랜잭션을 발생시킵니다. gas limit은 위와 같이 할당합니다. 너무 적게 넣을 경우 트랜잭션이 발생하지 않을 수 있습니다. 여러 값을 삽입해본 결과 해당 가스비용이 적정한 값으로 판단했습니다. 그리고 smart contract의 insertData라는 함수를 이용해서 contract에 값을 넣어주고 블록이 채굴될 때 까지 대기합니다.

그 아래의 코드는 방금 삽입한 메타데이터가 smart contract 배열의 몇 번째 인덱스에 삽입되었는지 확인하고 제대로 삽입되었는지 인덱스를 이용해서 확인하는 과정입니다.

위의 코드까지가 영상 clip을 받아와서 암호화 처리 후 IPFS에 저장, 메타데이터화, 데이터베이스 저장, smart contract 저장 까지의 한 사이클입니다.

이후에는 Smart contract에 저장된 메타데이터를 서버의 DataBase에 저장하는 내용을 살펴보겠습니다.

```
1  pragma solidity ^0.4.19;
2
3  contract MetaData {
4
5      struct UserStruct {
6          string userEmail;
7          uint userAge;
8          uint index;
9      }
10
11     mapping(uint => UserStruct) userStructs;
12     string[] private data;
13
14
15     function insertData(
16         string _dbData)
17         public
18     {
19         //if(isUser(userAddress)) revert();
20         data.push(_dbData);
21     }
22
23
24     function getData(uint id)
25         public
26         constant
27         returns(string dbData)
28     {
29         return(data[id]);
30     }
31
32     function getIndex()
33         public
34         constant
35         returns(uint count)
36     {
37         return data.length - 1;
38     }
```

Smart contract 에 저장된 메타데이터를 가져와서 서버의 데이터베이스 테이블에 저장하는 코드를 설명드리기 전에 먼저 Smart contract 코드부터 설명드리겠습니다. 위의 코드는 solidity 언어로 작성되었습니다(Smart contract 코딩은 solidity 가 표준이라고 생각하시면 됩니다.) 맨 윗줄의 pragma 부분은 solidity compiler 버전을 의미합니다. 그리고 contract MetaData{~}로 시작하는데 contract 는 다른 언어의 Class 정도로 이해하시면 될 것 같습니다.

contract 내부를 보시면 구조체 형태의 UserStruct 부터 마지막 function getIndex()까지 정의가 되어있습니다. 구조체 형태는 여기서는 정의만 하고 사용되지 않았습니다. 즉 아무런 의미가 없으므로 신경쓰시지 않아도 됩니다. (이전에 contract 작성시 테스트용으로 혹시 몰라서 만들어놨습니다)

그 아래의 mapping 형도 구조체를 저장하기위한 스토리지로써, 현재는 큰 의미가 없습니다. 실제로 필요한 부분은 그 아래의 코드들입니다.

data 라는 이름을 갖는 동적배열 string[]을 정의합니다. 메타데이터가 저장될 배열입니다. 즉 배열의 index 0 : '메타데이터' 형태로 저장된다고 이해하시면 됩니다.

아래의 function insertData() 함수가 data 배열에 메타데이터를 넣어주는 역할을 합니다. 즉, 파이썬 코드에서 inserData()함수를 호출해서 메타데이터를 넣어주면 smart contract 에 해당 메타데이터가 삽입되게 됩니다.

아래의 getData() 함수는 배열의 index 를 인자로 받고, 받은 index 에 해당하는 메타데이터를 반환해줍니다.

그리고 그 밑의 getIndex()가 현재 배열의 index 의 번호를 반환해줍니다. 이를테면 지금까지 저장된 메타데이터의 개수를 알고싶다면 getIndex()함수를 호출하고, 특정 메타데이터를 가져오고 싶다면 해당 메타데이터의 index 를 getData(index) 형식으로 확인할 수 있습니다.

하지만 이렇게 직접 smart contract 에 접근하는 방식은 상당히 비효율적입니다. 따라서 따로 서버의 DataBase 에 위 contract 배열의 index 와 메타데이터를 저장하게끔 구현했습니다. 아래에서 확인하실 수 있습니다.

```

rpc_url = "http://192.168.1.2:8545"           #Mac mini의 geth와 통신하기 위한 HTTPProvider
w3 = Web3(HTTPProvider(rpc_url))
contract = w3.eth.contract(abi=[{"constant": True, "inputs": [{"name": "id", "type": "uint256"}], "name": "getData", "outputs": [{"name": "dbData", "type": "uint256"}]}, {"payable": False, "stateMutability": "view", "type": "function"}])

conn = pymysql.connect(host='localhost', user='root', password='111111', db='blockChain', charset='utf8')

curs = conn.cursor()

tx_receipt = w3.eth.getTransactionReceipt('0xd501b20ee29b361f7babde65bbc78a13b583e8936a09aa235cde71289c39ebdb')
contract_address = tx_receipt['contractAddress']
contract_instance = contract(contract_address)

while True :
    sql = 'SELECT _id FROM metaData ORDER BY _id DESC LIMIT 1;'
    curs.execute(sql)
    temp_index = curs.fetchone()      #현재 DB에 저장된 마지막 열의 index를 가져온다.
    if temp_index is None :          #데이터베이스에 저장된 내용이 아무것도 없다면 실행
        f_values = contract_instance.call().getData(0)
        curs.execute("""INSERT INTO metaData VALUES(%s,%s,%s,%s,%s)""" , (0,f_values[1],f_values[2],f_values[3],f_values[4],f_values[5]))
        conn.commit()
        continue
    temp_index = temp_index[0]       #테이블의 index를 정수형으로 가져옴(튜플로 변환되기 때문)
    contract_index = int(contract_instance.call().getIndex())      #현재 컨트랙트에 저장된 마지막 배열의 index
    while contract_index > temp_index : #컨트랙트 배열의 index가 테이블의 index보다 크다면 컨트랙트에 저장된 데이터를 데이터베이스에 저장
        temp_index += 1
        contract_value = contract_instance.call().getData(temp_index)
        db_value = eval(contract_value)
        curs.execute("""INSERT INTO metaData VALUES(%s,%s,%s,%s,%s)""" , (temp_index, db_value[1], db_value[2], db_value[3], db_value[4], db_value[5]))
        conn.commit()
        print('inserted!')
    time.sleep(180)

```

Smart Contract 에 직접 Access 해서 메타데이터를 가져오는데에는 위에서 언급한 것처럼 비효율적일 수 있습니다.

이를 위해서 Server 에서 Smart contract 에 저장된 모든 메타데이터를 칼럼별로 구분하여 테이블에 저장하게끔 구현했습니다.

while 문 전까지는 이전에 설명했던 contract 와 통신하는 코드와 똑같습니다. 다른부분은 mysql 을 사용하기 위해 DB 서버와 연결하고 커서 객체를 정의하는 부분입니다.

While 문에서는 먼저 smart contract 에서 가져온 메타데이터가 저장된 metaData 테이블의 제일 마지막에 저장된 index 를 가져옵니다. 만일 현재 테이블이 빈 테이블(smart contract 에서 가져온 값이 아무것도 없음)이라면 smart contract 배열에 저장된 첫 번째 메타데이터 (index 0 에 저장된 메타데이터)를 가져와서 테이블에 저장하고 다시 반복문의 처음으로 돌아갑니다.(continue) 이제 빈 테이블이 아니므로 테이블에 마지막으로 저장된 index(이전과 같은 상황이라면 0)를 가져옵니다. 이후 가져온 index 의 값을 정수형으로 변환한 뒤 smart contract 의

배열 index 와 비교하기 위해 smart contract 배열의 마지막으로 저장된 index 를 contract_index 라는 변수에 저장합니다.

그 밑의 while 문으로 contract 에 저장된 값들을 데이터베이스에 삽입하는 과정입니다.

temp_index(데이터베이스의 metaData 테이블에서 마지막으로 저장된 index)와 contract_index(smart contract에 마지막으로 저장된 index)를 비교합니다.

목표는 smart contract 에 저장된 모든 메타데이터를 데이터베이스 metaData 테이블에 저장하는 것입니다.

따라서 temp_index 가 contract_index 보다 작다면 테이블에 저장해야하는 데이터가 아직 남아있다는 의미 이므로 temp_index 와 contract_index 가 같아질 때 까지 메타데이터를 smart contract 로부터 가져와서 테이블에 삽입하여 줍니다.

만일 모두 삽입이 되어서 temp_index 와 contract_index 의 값이 같아졌다면 3 분간 대기합니다. 그 사이에 smart contract 에 메타데이터가 삽입이 된다면 당연히 contract_index 의 값도 증가할 것이므로 3 분 뒤에 temp_index 의 값보다 커질것이고 다시 위와 같은 과정을 통해 metaData 테이블과 smart contract 의 메타데이터를 동기화 시켜줍니다.

-그 외 issue

위에서 말씀드린 내용들이 현재 프로젝트에서 사용되는 코드 및 기능들입니다. 최대한 예외처리를 통해서 자잘한 프로그램 에러(ipfs, openRTSP, geth)를 잡으려고 노력했지만 분명 예기치 않은 에러가 생길 때가 있습니다.

제가 겪은 시행착오들을 최대한 설명드리기 위해서 따로 항목을 만들었습니다.

1. IPFS 관련

프로그램을 수정하시거나 실행하시거나 개발하시는 과정에서 분명 IPFS 와 관련된 이슈가 많습니다.

가장 중요한것은 프로그램을 실행하는 로컬 repo(IPFS block 들이 저장되고 node id, configure file 등이 존재)에 hash 가 존재하지 않으면 머클디렉토리를 구성하거나 ipfs get 을 하는데에 있어 시간이 오래걸릴 수 있다는 점입니다.

- ipfs api not running 관련

위에서도 언급을 했지만 프로그램 실행중에 발생하는 대표적인 ipfs 버그(에러)입니다.

위의 에러가 발생하면 ipfs repo fsck 를 입력한 뒤 ipfs daemon 을 다시 구동시키셔야 합니다.

macbook 에서는 이러한 현상이 거의 없고 raspberry pi 에서만 나타나는걸 보면 아무래도 스펙 문제와 관련이 있는것 같습니다.

해당 에러를 처리하기 위해 예외처리 구문을 작성했지만 완벽하지 않습니다. 해당 에러 자체가 ipfs 자체 bug 이기도 하고 발생하는 원인이 한 두가지가 아니기 때문입니다.

2.geth 관련

geth 는 동기화 하는데에 많은 프로세스 파워가 필요합니다.

따라서 라즈베리파이와 같은 미니 컴퓨터에서는 많이 버거워하는게 사실입니다.

이를 위해서 따로 B.C node(blockchain node)를 구축한 것이고 B.C node 에서만 geth 동기화를 진행, 라즈베리파이가 해당 B.C node 의 chain-data 를 이용해서 blockchain 과 트랜잭션을 주고받을 수 있습니다.

한 가지 주의하실 점은 B.C node 에서 geth 동기화만 시킨다고 트랜잭션이 가능한게 아닙니다. 지갑 계정, 즉 eth.accounts[0]의 lock 을 해제시켜주어야 합니다.

이를 위해서는 geth 를 동기화 하고 있는 B.C node(Mac mini)의 콘솔 창에서 geth attach <http://127.0.0.1:8545> 를 입력하면 자바스크립트 대화형 콘솔로 진입합니다.

```
web3.personal.unlockAccount(eth.accounts[0], "gksmf5081", 0)
```

을 입력해서 True 가 반환되면 성공입니다.

eth.accounts[0]은 lock 을 풀어주려는 account이고

“gksmf5081”은 해당 계정의 비밀번호, 0 은 geth 가 동기화하는 동안 lock 을 계속 해제하고 있겠다는 의미입니다.

-참고 사이트

개발을 하는데에 있어 제가 참고했던 자료 및 사이트를 기재하겠습니다.

1. [IPFS 명령어 옵션](#)
2. [Geth 명령어 옵션](#)
3. [openRTSP 명령어 옵션](#)
4. [Web3.js를 이용한 스마트컨트랙트 기초 강좌\(나도 Dapp 개발\)](#)
5. [Python\(web3.py\)를 이용한 스마트컨트랙트 기초 강좌](#)