# Chapter 4. 이더리움 클라이언트 Geth 명령어들

Geth는 블록체인에 대해 명령어를 입력하고 결과를 받아볼 수 있는 클라이언트이다. 이 콘솔창에서 eth, admin, miner, personal 모듈의 명령어를 배우게 된다. 블록체인에서 사용하게 되는 계정과 주소를 생성할 수 있게 되고, 디지털 서명, 해싱을 이해하게 된다.

# 1. Geth 모듈 명령어들

Geth는 **G**o 프로그래밍 언어로 구현된 **eth**ereum이라는 앞 글자를 따와서 이름이 붙여졌다. 앞서 배웠듯이, Geth를 실행하면 네트워크에 접속이 되고 현재 사용하는 컴퓨터를 이더리움 노드로 만든다.

그렇다고 Geth가 접속 도구로만 사용되는 것은 아니다. Geth는 연결한 블록체인에 대한 클라이언트 역할도 한다.

블록체인을 구성하는 분산시스템에는 서버라는 개념이 없지만, 블록체인을 서버로 가정하고 내 컴퓨터가 그 서버에 대한 클라이언트로 생각하면 이해가 쉬울 수 있다. 단 서버는 서비스를 제공하고, 클라이언트는 이를 요청하는 그런 주종 관계가 아니라는 것은 분명히 이해하자. 분산시스템에서는 참여하는 모든 노드가 서버가 될 수 있기 때문이다.

Geth 콘솔을 열어서 자바스크립트 프로그래밍을 할 수 있는데, 그 밖에도 블록체인에 프로그램을 배포하거나, 마이닝하거나, 상태를 파악하는 등 블록체인에 대한 여러 기능을 활용할 수 있다.

단말을 두 개 열어서 사용하자. 하나는 \_gethNow를 실행하여 콘솔없는 geth를 띄운다 (명령창B라고 하자). 또 다른 단말은 geth attach http://localhost:8445로 콘솔있는 geth를 열어놓는다 (명령창A라고 하자). 앞으로 > 프롬프트는 명령창A에서의 작업이다.

여기서는 (1) eth, (2) admin, (3) miner, (4) personal 4개의 모듈에서 제공하는 명령어를 배워보자. 명령어는 대문자와 소문자를 구별하므로 정확하게 입력해야 한다.

## 1.1 eth 명령어

명령창A에서 "eth."를 입력한 후 키를 눌러보면 (> eth.<TAB>), eth 모듈에서 제공하는 API를 출력한다. 계정, 블록, gas, 트랜잭션, 디지털 사인 관련 명령을 제공한다.

명령창A의 pjt\_dir> geth attach http://localhost:8445  
geth> eth.<TAB>  
eth.\_requestManager eth.getAccounts eth.getProtocolVersion eth.isSyncing  
eth.accounts eth.getBalance eth.getRawTransaction eth.mining  
eth.blockNumber eth.getBlock eth.getRawTransactionFromBlock eth.namereg  
eth.call eth.getBlockNumber eth.getStorageAt eth.pendingTransactions  
eth.chainId eth.getBlockTransactionCount eth.getSyncing eth.protocolVersion  
eth.coinbase eth.getBlockUncleCount eth.getTransaction eth.resend  
eth.compile eth.getCode eth.getTransactionCount eth.sendIBANTransaction  
eth.constructor eth.getCoinbase eth.getTransactionFromBlock eth.sendRawTransaction  
eth.contract eth.getCompilers eth.getTransactionReceipt eth.sendTransaction  
eth.defaultAccount eth.getGasPrice eth.getUncle eth.sign  
eth.defaultBlock eth.getHashrate eth.getWork eth.signTransaction  
eth.estimateGas eth.getMining eth.hashrate eth.submitTransaction  
eth.filter eth.getPendingTransactions eth.iban eth.submitWork  
eth.gasPrice eth.getProof eth.icapNamereg eth.syncing

### 계정 정보

3장에서 생성한 계정을 명령창A에서 출력해보자. 물론 계정은 여러 개 생성할 수 있고, 생성된 순서로 인덱스가 주어지고, 그 인덱스를 지정하면 특정 account를 조회할 수 있다.

> eth.accounts  
["0xc8ea4c4e655f8152adc075a649aa7ec35564c7c0", "0x4fa2c7caac80a8518264d263bdb5ed74f1a6f398"]  
> eth.accounts[0]  
"0xc8ea4c4e655f8152adc075a649aa7ec35564c7c0"

### 계정 잔고

계속해서 명령창A에서 계정의 잔고를 알아보자. 계좌를 새로 개설했으므로 잔고는 없다고 출력한다.

> eth.getBalance(eth.accounts[0]);  
0  
> eth.getBalance(eth.accounts[1]);  
0

계좌에 잔고가 없으면 거래를 하거나, 프로그램을 블록체인에 올리거나 어떤 일도 할 수 없다.

마이닝을 통해서 충전을 해보자. miner.start()를 실행하고, 또 다른 백그라운드 명령창B에서는 어떤 변화가 있는지 살펴보자. 수 분만에 다음과 같이 5 ether가 입금되었고, 그 금액은 마이닝이 실행되면서 늘어나고 있다. 어느 정도 충전이 되었으면 miner.stop()으로 중단하기로 한다.

main, testnet에서는 마이닝이 언제나 실행되고 있으므로 별도의 마이닝이 필요없겠지만, 사설망에서는 컴퓨터가 마이닝 작업으로 인해 느려지는 것을 막기 위해서 필요한 경우에만 마이닝을 켜고 끌 수가 있다.

> miner.start();  
null # null이 반환된다고 오류가 아니다.  
> eth.getBalance(eth.accounts[0]); # 시간이 지나면서 충전이 증가한다.  
5000000000000000000  
> eth.getBalance(eth.accounts[0]);  
60000000000000000000  
> eth.getBalance(eth.accounts[0]);  
80000000000000000000  
> eth.getBalance(eth.accounts[0]);  
285000000000000000000  
> miner.stop();  
null

앞서 충전된 금액의 단위는 Wei이다. 이를 ether로 변환해보자.

> bal = eth.getBalance(eth.accounts[0])  
285000000000000000000  
> web3.fromWei(bal,"ether")  
285

### 블록 갯수

블록이 몇 개나 있는지 알아보자. 독자가 사용하는 컴퓨터에서는 다른 값이 나올 수 있겠지만, 여기서는 57개가 생성되었다고 출력되었다. 블록이 생성이 되었다는 것은 마이닝이 그만큼 완성되었다는 의미이고, 그 보상이 계좌에 입금된 것이다.

coinbase 또는 etherbase는 입출금이 일어나는 자신의 주 계정을 말한다.

> eth.blockNumber  
57  
> eth.coinbase  
"0xc8ea4c4e655f8152adc075a649aa7ec35564c7c0"

지금까지는 geth 콘솔 프롬프트 >에서 작업하는 것을 보았다.

geth 콘솔에서 한 줄씩 입력했던 내용은 사실 자바스크립트 코드였다. 이 명령문들을 묶어 저장해서 일괄 실행할 수 있다.

[프로그램] src/ethCommands.js  
var primary = eth.accounts[0];  
var bal=eth.getBalance(primary);  
console.log('balance in Wei: ', bal);  
console.log('balance in ether: ', web3.fromWei(bal, "ether"));  
console.log('blocknumber: ', eth.blockNumber);  
console.log('coinbase: ', eth.coinbase);

위 자바스크립트 파일을 실행하려면 loadScript() 명령어를 사용한다.

주의: **loadScript() 따옴표**

유닉스계열에서는 따옴표를 구분할 필요 없지만, 윈도우에서는 큰 따옴표를 밖에, 작은 따옴표를 안에 쓴다. 또한 윈도우 운영체제에서 실행한다고 하더라도, 유닉스의 디렉토리 구분자 (/)를 사용한다.

pjt\_dir> geth --exec "loadScript('src/ethCommands.js')" attach http://localhost:8445

또 다른 명령창을 하나 열어서 실행을 한다. 단, 현재 프로젝트 디렉토리에서 실행하고 "ethCommands.js"파일은 프로젝트의 서브 디렉토리인 src에 있다고 가정하자. "--jspath" 옵션을 지정해서 자바스크립트 파일이 있는 디렉토리를 지정할 수 있다.

pjt\_dir> geth --jspath "src" --exec "loadScript('ethCommands.js')" attach http://localhost:8445  
balance in Wei: 100000000000000000000  
balance in ether: 100  
blocknumber: 0  
coinbase: 0xc8ea4c4e655f8152adc075a649aa7ec35564c7c0

## 1.2 admin 명령어

admin모듈은 Peer, RPC 등 노드 관리에 필요한 명령어를 제공한다. 마찬가지로 geth 프롬프트 >에서 admin.하고 을 누르면 제공되는 admin의 API를 볼 수 있다.

> admin.<TAB>키를 누른다.  
admin.addPeer admin.getDatadir admin.nodeInfo admin.sleepBlocks admin.toString  
admin.addTrustedPeer admin.getNodeInfo admin.peers admin.startRPC admin.valueOf  
admin.clearHistory admin.getPeers admin.propertyIsEnumerable admin.startWS  
admin.constructor admin.hasOwnProperty admin.removePeer admin.stopRPC  
admin.datadir admin.importChain admin.removeTrustedPeer admin.stopWS  
admin.exportChain admin.isPrototypeOf admin.sleep admin.toLocaleString

admin.nodeInfo 명령어는 node 자신과 관련한 정보를 알려 준다.

> admin.nodeInfo  
{  
 enode: "enode://94b49...생략...76963@xxx.xxx.xxx.xxx:38445",  
 enr: "0xf89cb...생략...2962d",  
 id: "a49c9...생략...2e00e",  
 ip: "125.176.xxx.xxx",  
 listenAddr: "[::]:38445",  
 name: "Geth/jsl/v1.8.20-stable-24d727b6/windows-amd64/go1.11.2",  
 ports: {  
 discovery: 38445,  
 listener: 38445  
 },  
 protocols: { 여기서는 genesis.json에서 설정한 정보를 출력하고 있다.  
 eth: {  
 config: {   
 chainId: 33,  
 eip150Hash: "0x0000000000000000000000000000000000000000000000000000000000000000",  
 eip155Block: 0,  
 eip158Block: 0,  
 homesteadBlock: 0  
 },  
 difficulty: 2349526312,  
 genesis: "0x5704d029fe80f4fb605c0cb5e31d591511f10a46a0cb8166f97d8d559f9bc5b0",  
 head: "0x83879983f659150f680b796c1570efed581ab180483a12277b35e845e71b4830",  
 network: 33  
 }  
 }  
}  
>

블록체인에 연결되어 있는 다른 컴퓨터(peer)가 있다면 출력할 수 있다. 프롬프트에서 "admin.peers" 라고 실행시킬 수도 있지만, 명령행(command line)에서 geth로 다음과 같이 실행하고 출력할 수도 있다.

pjt\_dir> geth --exec admin.peers attach http://localhost:8445  
[{  
 caps: ["eth/62", "eth/63"],  
 enode: "enode://a8728...생략...b1a0f@159.89.156.144:30303",  
 id: "1f93ed90d40989a95e3460ab8a9dc2a2d4660f0b368e36f3a582ea3932ed1e8c",  
 name: "Geth/v1.8.10-stable-eae63c51/linux-amd64/go1.10.1",  
 network: {  
 inbound: false,  
 localAddress: "117.xxx.xxx.xxx:35582",  
 remoteAddress: "159.89.xxx.xxx:30303",  
 static: false,  
 trusted: false  
 },  
 protocols: {  
 eth: "handshake"  
 }  
}]

### enode는 고유주소

**enode**는 이더리움 노드를 표현하는 URL이다. 따라서 각 노드의 enode는 모두 서로 다르다.

우리가 사용하는 URL과 같이 고유의 ID로서, enode://<<사용자이름>>@<<호스트>>:<<포트>>으로 구성된다.

**사용자 이름**은 64바이트 공개키, **호스트**는 도메인명이 허용되지 않으며 IP주소로 나타낸다.

### enode는 통신에 사용된다

고유한 enode를 통해 서로 인식하고 통신할 수 있다. 사설망에서 다른 컴퓨터를 참여시키고 싶을 때 enode를 넣어주며, 그렇게 참여가 되면 사설망에서도 다른 컴퓨터의 계좌에 송금을 하거나 거래를 할 수 있게 된다.

### peer 발견

이더리움에서는 peer를 발견하는 방법으로 Kademlia라고 하는 알고리듬의 일부를 수정해서 적용한다.

사설망에서 이러한 알고리듬에 따라 자동으로 잘 인식하지 못할 경우, 다음 방법으로 추가할 수 있다고 설명하였다. 그러면 지정한 enode를 통해 peer를 인식한다.

* 1. geth --bootnodes 명령어로 시작할 때부터 peer의 enode를 적어두거나,
  2. /static-nodes.json 파일에 enode를 적어서 저장해두거나,
  3. admin.addPeer("enode://f4642...416c0@33.4.2.1:30303") 함수에 적어두거나 한다.

### peer 상황 확인

net.peerCount 명령어로 현재 peer 노드의 수를 확인할 수 있다. 현재 0이지만 걱정할 필요는 없다. peer의 수는 참여했다가 안했다가 할 수 있으므로 가변적일 수 밖에 없다. net.listening으로 peer를 찾고 있는 상태인지 확인할 수 있다.

"Looking for peers" 이런 메시지가 계속 뜨는 경우는 Geth에서 peer를 찾는 과정이다. --nodiscover를 옵션에 넣으면 그 메시지가 사라지게 된다.

그렇다고 해서 문제가 되는 것은 아니다. 자신이 다른 peer를 찾지 않을 뿐, 다른 peer가 자신의 노드에 접속하지 못하는 것은 아니다.

> console.log('peer count: ', net.peerCount);  
peer count: 0  
> console.log('net.listening: ', net.listening);  
net.listening: true

## 1.3 miner 명령어

역시 geth 프롬프트 >에 "miner. "을 입력하면, miner의 API를 출력할 수 있다. miner모듈은 마이닝을 시작하거나 종료할 수 있는 명령어와 마이닝 관련 etherBase, gasPrice를 설정할 수 있다.

> miner.<TAB>  
miner.constructor miner.isPrototypeOf miner.setExtra miner.start miner.toString  
miner.getHashrate miner.propertyIsEnumerable miner.setGasPrice miner.stop miner.valueOf  
miner.hasOwnProperty miner.setEtherbase miner.setRecommitInterval miner.toLocaleString

### 마이닝 속도

**마이닝은 Target Hash를 맞출 때까지 계속 해시(Hash)를 생성하여 시도하는 과정**이다.

마치 숫자 맞추기 게임에서, 상대가 숫자를 하나 생각하고 있고, 그 수를 맞추기 위해 계속 말한다. 비유하면 1초에 몇 번이나 그 시도를 했는지가 hash rate이다. 마이닝하는 속도를 의미한다. **hash rate**은 **1초에 시도한 횟수**, 즉 h/s (h: 해시의 갯수, s: 초)이다.

결과로 0이 출력될 수 있다. 그렇다면 마이닝도 했는데, 해시가 전혀 생성되지 않았다는 의미일까?

그렇지 않다. 마이닝을 하면 그 값을 볼 수 있다. 나중에 마이닝에 대해서 배우겠지만, hashrate은 마이닝을 하면서 발생하는 지표이기 때문이다.

> eth.hashrate 마이닝을 하기 전, hashrate은 0이다.  
0  
> miner.start() 마이닝을 시작하고, 백그라운드에서 망치 아이콘과 함께 mined potential block이 출력된다.  
null  
> eth.hashrate 마이닝을 시작하자, hashrate가 집계되기 시작한다.  
83522  
> eth.hashrate 마이닝을 켜놓은 상태이고, hashrate가 증가하고 있다.  
193285  
> miner.stop() 마이닝을 정지한다  
null  
> eth.hashrate 마이닝을 정지하니까, hashrate이 감소한다.  
65930

### 지급계좌 변경

**etherBase**는 마이닝이 성공하면 보상이 주어진다. 그 입금 계좌를 말한다.

경우에 따라서, coinbase를 변경해야 할 때가 있다. 그럴 때에는 miner.setEtherbase 명령어로 교체할 계좌를 넣어주면 된다.

현재 coinbase를 accounts[1]로 변경해보자. 이번에는 파일에 명령어를 입력하고 실행해 보자.

[프로그램: src/ethMinerEtherbase.js]  
console.log('Before: ', eth.coinbase);  
miner.setEtherbase(eth.accounts[1]);  
console.log('After: ', eth.coinbase);

pjt\_dir> geth --exec "loadScript('src/ethMinerEtherbase.js')" attach http://localhost:8445  
Before: 0x21c704354d07f804bab01894e8b4eb4e0eba7451  
After: 0x778ea91cb0d0879c22ca20c5aea6fbf8cbeed480

## 1.5 personal 명령어

계정 및 지급과 관련한 명령어를 포함하고 있으므로, RPC로 사용할 경우 주의가 필요하다.

> personal.<TAB>  
personal.\_requestManager personal.getListAccounts personal.listWallets personal.sendTransaction  
personal.constructor personal.getListWallets personal.lockAccount personal.sign  
personal.deriveAccount personal.importRawKey personal.newAccount personal.signTransaction  
personal.ecRecover personal.listAccounts personal.openWallet personal.unlockAccount

### 계정 목록

자신이 개설한 계정의 목록을 확인할 수 있다. 자신이 몇 개를 개설했는지에 따라 계좌번호들이 출력된다.

> personal.listAccounts  
["0xc8ea4c4e655f8152adc075a649aa7ec35564c7c0", ...생략..., "0x6f94a3529222ad7b67305c30928df0fd2d70966d"]

### 거래 대기

블록체인에 거래가 발생하면, 그 거래가 실시간으로 처리되지 않는다. 거래가 여러 건 발생하면, 어느 거래는 처리되고 그렇지 않은 경우에는 대기상태가 된다.

거래가 마이닝되기 전까지 transaction pool에 남겨지게 된다. txpool은 트랜잭션이 마이닝되기 전에 대기하는 transaction pool의 상황을 알려 준다.

txpool.status는 갯수로 알려주고 있다. 마이닝이 적체되어 있는 경우, 그 상황을 알 수 있어 사설망에서는 매우 요긴하게 쓰인다.

> txpool.inspect  
{  
 pending: {},  
 queued: {}  
}  
> txpool.status  
{  
 pending: 0,  
 queued: 0  
}

# 2. 계정

계정은 은행의 계좌라고 생각하면 쉽게 이해된다. 이더리움의 계정은 은행 계좌처럼 잔고를 가지고 있어 수금, 송금이 가능하다.

계정은 쉽게 만들 수 있다. 그러나 우리가 사용하는 은행 계좌와 달리 암호화되어 있기 때문에 단순하지 않고, 숫자와 문자가 섞인 난해하게 생긴 문자열로 이루어진다.

계정은 온라인에 연결되어 있지 않아도 만들 수 있다. 계정은 거래가 최초 발생해야 비로서 블록체인에 기록이 되고 알려진다.

누구나 그 계정에 입금을 할 수 있다. 존재하지 않거나, 인증할 수 없는 계정으로의 입금은 거래가 성립할 수 없다.

출금은 당연히 **디지털 서명**이 되어야 한다. 은행에서 입출금 거래를 하면서 서명 또는 날인을 하듯이 블록체인 거래도 그러하다. **디지털 서명**이 되어 있어야 거래로 인정된다.

거래가 위변조되면 디지털 서명이 맞지 않게 된다. 이 경우 새로 서명을 해야 한다. 누군가 악의적으로 원래의 디지털 서명을 사용하지 않으면, 대칭키인 공개키(public key)로 풀 수 없게 된다.

## 2.1 계정의 종류

이더리움에서 계정은 블록체인 안쪽 또는 바깥쪽에 있는지를 기준으로 **외부 계정** 또는 **내부 계정**으로 구분할 수 있다.

|  |  |
| --- | --- |
| 구분 | 설명 |
| 외부 계정 | 블록체인 외부의 계정, 예를 들면, 우리가 사용하는 계정을 말한다. |
| 내부 계정 | 블록체인 내부의 계정, 보통 스마트 계약의 계정을 말한다. |

### 외부 계정

블록체인 **외부 액터가 가지는 계정(Externally Owned Account)**을 말한다. 외부 액터는 보통 개인이지만 법인도 물론 될 수 있다. 그 계정은 컴퓨터, 인터넷 또는 하드웨어에 존재할 수도 있다.

외부 계정은 고유의 **키**를 가지고 있어 트랜잭션을 **사인(sign)**할 수 있다. 어떤 물품을 구매한다면 계정에 있는 잔고에서 디지털사인을 하고 판매자에게 송금을 하게 된다.

### 계약 계정

**블록체인 내부의 자율 객체(Autonomous Object)가 가지는 계정**을 말한다. 자율 객체란 고유의 주소와 상태정보를 가지고 블록체인 안에 가상으로 존재한다. 보통은 **프로그램 코드를 가지고 실행되는 스마트 계약**을 자율 객체라고 말한다.

스마트 계약의 **프로그램 코드가 배포되고 계정이 생성**되며, 그 순간부터 상태정보를 가질 수 있게 된다.

계약 계정이 특별난 것은 **잔고**를 가지기 때문이다. 프로그램 코드에 의해 실행되는 계약 계정이 주소를 가질 수 있고 잔고를 가질 수 있기 때문에 코인을 발행할 수 있다. 그러나 **개인 계정과 달리 키가 없어** 트랜잭션에 사인할 수는 없다.

## 2.2 키와 주소 생성

이더리움은 한 계정에 하나의 주소를 가진다. 비유하면, 계정은 통장으로, 주소는 통장의 계좌번호와 비슷하다고 볼 수 있다.

다음에서 보듯이 geth account new 명령어로 계정을 생성하면 주소 역시 생성된다.

주소와 계정

**주소 address**는 **20 바이트 코드**로 계정 account의 고유 번호이다. 다른 말로 계정은 하나의 주소 식별자를 가진다. 계정은 잔고와 거래 건수를 가진다. 계정은 외부 계정 또는 컨트랙이 가질 수 있다.

### 주소 생성 절차

주소는 개인키에서 생성된 공개키로부터 생성된다. 그 순서는:

문자열 해싱 --> 개인키 --(ECDSA)--> 공개키 --(Keccak-256 해싱, BASE58Chcek)--> 주소

### 개인키 생성

개인키는 무작위로 생성된다. 비유를 하면 동전을 256번 던져서 256비트의 수를 생성하는 것과 같다. 따라서 이 수를 우연히 맞추는 확률은 사실상 불가능하다.

개인키는 16진수로 저장되어 있다. hexadecimal 16진수 64자리, 즉 32바이트 길이로 생성된다.

기억이 난다면, keystore에 저장된 "UTC--"로 시작하는 키가 그것이다. 이 키를 우리가 직접 보는 경우는 드물다.

이 키는 컴퓨터가 사용하게 되고, 사용자는 나중에 배우는 WIF(Wallet Import Format) 형식의 주소라고 하는 것을 보게 된다.

3a1076bf45ab87712ad64ccb3b10217737f7faacbf2872e88fdd9a537d8fe266  
 1 2 3 4 5 6 (길이를 10자리마다 표시)

16진수

수가 커지면 2진수 보다 16진수로 표현하는 것이 편리하다. 6을 의미하는 hexa-, 10을 의미하는 decimal이 합쳐져서 **hexadecimal**은 16진수를 의미하고 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f로 세어간다. 16진수는 1자리를 **니블(nibble)**이라고 하며, 16까지 표현하므로 4비트를 차지하게 된다. 즉 1 니블의 16진수 = 4 bits = 1/2 byte **2 니블이 1바이트**가 된다. 16진수로 식별하기 위해 앞에 **0x**를 붙일 수 있다.

파이썬으로 키와 주소를 생성해 보자. 여기서는 비트코인에 대하여 키, 주소, 사인 등을 해보는데, 이더리움과 큰 차이가 없다.

이더리움의 Vitalik Buterin이 개발한 비트코인 라이브러리인 **Pybitcointools**는 더 이상 유지 보수하지 않겠다고 했고, 그 후 다른 라이브러리에서 이어지고 있다. 예를 들어 **Pycryptotools** ( https://github.com/primal100/pybitcointools )이 그 라이브러리를 다른 암호화폐도 지원하도록 확장하여 제공되고 있다. 이 라이브러리를 설치하려면 pip install crpytos라고 해준다(윈도우에서는 pip대신 pip3를 사용한다. https://pypi.org/project/cryptos/ ). 이 라이브러리는 bitcoin 뿐만 아니라 Litecoin, Dash, Dogecoin도 지원하고 있다.

또한 bitcoin (https://pypi.org/project/bitcoin/) 라이브러리도 이어서 개발되고 있다. 이 라이브러리는 비트코인에 한정하여 사용할 수 있다. 여기서는 **bitcoin** 라이브러리를 사용하기로 하고, 설치해 보자.

pip install bitcoin

cryptos나 bitcoin 라이브러리는 업그레이드가 몇 년이나 되지 않아서, **최신 버전의 파이썬과는 작동이 잘 되지 않는다**는 점을 주의한다. **현재 Python 3.7 버전**에서 하고 있다.

우선 개인키(private key)를 생성해보면, 16진수 64자리 **32 바이트**가 생성된다. 비트코인에서의 사적키는 이더리움과 마찬가지로 32 바이트 길이를 가지고 차이가 없다. 사적키는 무작위로 생성이 되기 때문에, 그냥 'hello key'라는 문자열을 사용하였다.

아래는 Python에서 실행하고 있다. Python 프롬프트는 geth의 그것과는 다르다는 점에 유의하자.

>>> import bitcoin  
>>> privKey = bitcoin.sha256('hello key')  
>>> print(privKey)  
3e295c8dc78fb7f3865067dfc42fe845263db7661296e7e32e3a37baa1a27a7b

hash 값을 구하는 것이기 때문에 다른 라이브러리 hashlib을 사용해도 동일한 hash를 얻을 수 있다.

>>> import hashlib  
>>> x=hashlib.sha256("hello key".encode('utf-8')).hexdigest()  
>>> print(x)  
3e295c8dc78fb7f3865067dfc42fe845263db7661296e7e32e3a37baa1a27a7b

### 개인키에서 공개키 생성

개인키로부터 **ECC(Elliptic Curve Crypotography)**에 따라 공개키를 생성한다.

ECC를 사용하는 ECDSA(Elliptic Curve Digital Signature Algorithm)는 많이 사용되고 있는 RSA (Rivest–Shamir–Adleman)에 비해 효율적이다. RSA가 128 바이트 키를 생성하는데 비해, ECDSA는 20 바이트로 보다 **짧은 키**로 **같은 수준의 보안**을 지킬 수 있다. 3072 비트 RSA key는 대략 256 비트 ECDSA이면 같은 수준의 보안을 지켜낼 수 있다. 여기서는 64비트짜리 키를 생성한다.

>>> pubKey = bitcoin.privtopub(privKey)  
>>> print(pubKey)  
04f6cc26cec156c180f8...생략...4e87cf98001d428f564f 비트코인 공개키는 65바이트

### 공개키에서 주소 생성

공개키 64바이트를 Keccak-256 hashing해서 32바이트로 생성한다 (당연히 256 해싱이니까 256 비트). 앞의 12바이트를 제거하고, 뒤 20 바이트를 주소로 쓰게된다. 그리고 hex코드를 사람이 읽을 수 있는 base58 코드로 변환해서 주소로 쓴다.

주소는 20 바이트 길이, 또는 아래와 같이 0x를 붙여 21 = 0x + 20 바이트이다. **대소문자는 체크섬(checksum)** 용도로 쓰이기 때문에 꼭 지켜야 한다.

0xC2D7CF95645D33006175B78989035C7c9061d3F9  
 1 2 3 4 (길이를 10자리마다 표시)

base58

이진화된 바이트 코드를 58개의 알파벳과 숫자로 구성된 문자열로 변환하는 것으로 비트코인 주소를 인코딩할 때 사용한다. 알파벳과 숫자 즉 영문 대소문자 52개와 1~9까지 총 61개 문자에서 혼돈할 수 있는 3개의 문자 (O, I, l)를 제외한다.

비트코인의 주소는 26-35 크기를 가지는데, 보통 34 글자로 이루어진다. 아래에서도 이를 확인할 수 있다. 이더리움의 주소는 20바이트, 비트코인은 17바이트의 길이를 가진다.

>>> addr = bitcoin.pubtoaddr(pubKey)  
>>> print(addr)  
1NthZ9kJVbtxrHQiocfjLLTcMH3F2DLcgD

### WIF

key는 16진수로 사용하거나 WIF(Wallet Import Format)형식으로 바꾸어 표현할 수도 있다.

WIF는 그냥 복잡해 보이는 key를 **복사하거나 다루기 편리하게 다른 형식으로 표기하는 것**일 뿐이다. 원하면, **생성했던 절차를 반대로 수행해서 원래의 개인키로 변환, 회복**될 수 있다.

개인키를 온라인에서 WIF변환하는 서비스가 있지만, 자신의 개인키를 함부로 온라인에 공유하지 않도록 한다.

우리가 보는 개인키는 이미 WIF형식으로 만들어져 있다. 즉 우리가 개인키를 원시 그대로 보려면, 이더리움에서는 별도로 작업을 해서 생성해야 한다.

개인키는 16진수로 저장되어 있다. 기억이 난다면, keystore에 저장된 "UTC--"로 시작하는 키가 있었다. 그 키를 읽고, 암호를 풀어내야 비로서 개인키를 볼 수 있다.

다음 ECDSA를 Wallet Import Format (WIF)으로 변환하는 과정을 예로 들어 보자 (참조: https://en.bitcoin.it/wiki/Wallet\_import\_format) WIF는 키가 짧아져서 복사할 때 오류가 적어지고 관리하기 편리하다.

Key를 base58Check 형식으로 변환하려면, 우선 버전 바이트를 맨 앞에 추가한다.

비트코인 주소는 0 (16진수로 0x00)이고 이더리움 메인네트워크는 128 (16진수 0x80)이다.

여기에 더블 해싱을 해준다. 32바이트에서 4바이트만 가져와 오류확인용 체크섬으로 뒤에 붙여준다. 그리고 base58변환을 해 주면 된다.

|  |  |  |
| --- | --- | --- |
| 단계 | 설명 | 예 |
| 1 | ECDSA 키를 생성 | 0C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D |
| 2 | 버전 1 바이트 추가 | 800C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D |
| 3 | SHA-256 hash | 8147786C4D15106333BF278D71DADAF1079EF2D2440A4DDE37D747DED5403592 |
| 4 | double hash 다시 한 번 더 SHA-256 hash | 507A5B8DFED0FC6FE8801743720CEDEC06AA5C6FCA72B07C49964492FB98A714 |
| 5 | 처음 4 bytes를 가지고 체크섬으로 사용 | 507A5B8D |
| 6 | 단계 2의 결과 맨 끝에 단계 5를 붙임 | 800C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D507A5B8D |
| 7 | base58로 WIF 변환 | 5HueCGU8rMjxEXxiPuD5BDku4MkFqeZyd4dZ1jvhTVqvbTLvyTJ |

버전정보

비트코인에서는 0x00, 테스트넷 0x6F, private key 0x80을 사용한다. 이더리움에서는 메인 네트워크에 0x80, 테스트네트워크에 0xef를 사용한다.

### 주소 파일

주소 파일은 이더리움 노드 data directory아래에 있다. 기억하겠지만 geth를 띄울 때 datadir 옵션에 해당하는 디렉토리를 말한다. 그 아래를 보면 키는 keystore 디렉토리 안에 UTC--<<년>>-<<월>>--주소와 같은 JSON 형식으로 저장되어 있다. 이 파일 안에는 private key, 비밀번호, 주소가 암호화되어 저장되어 있다.

아래는 리눅스에서 해당 파일의 내용을 출력하고 있다. 윈도우에서도 해당 디렉토리로 가서, 내용을 출력할 수 있다.

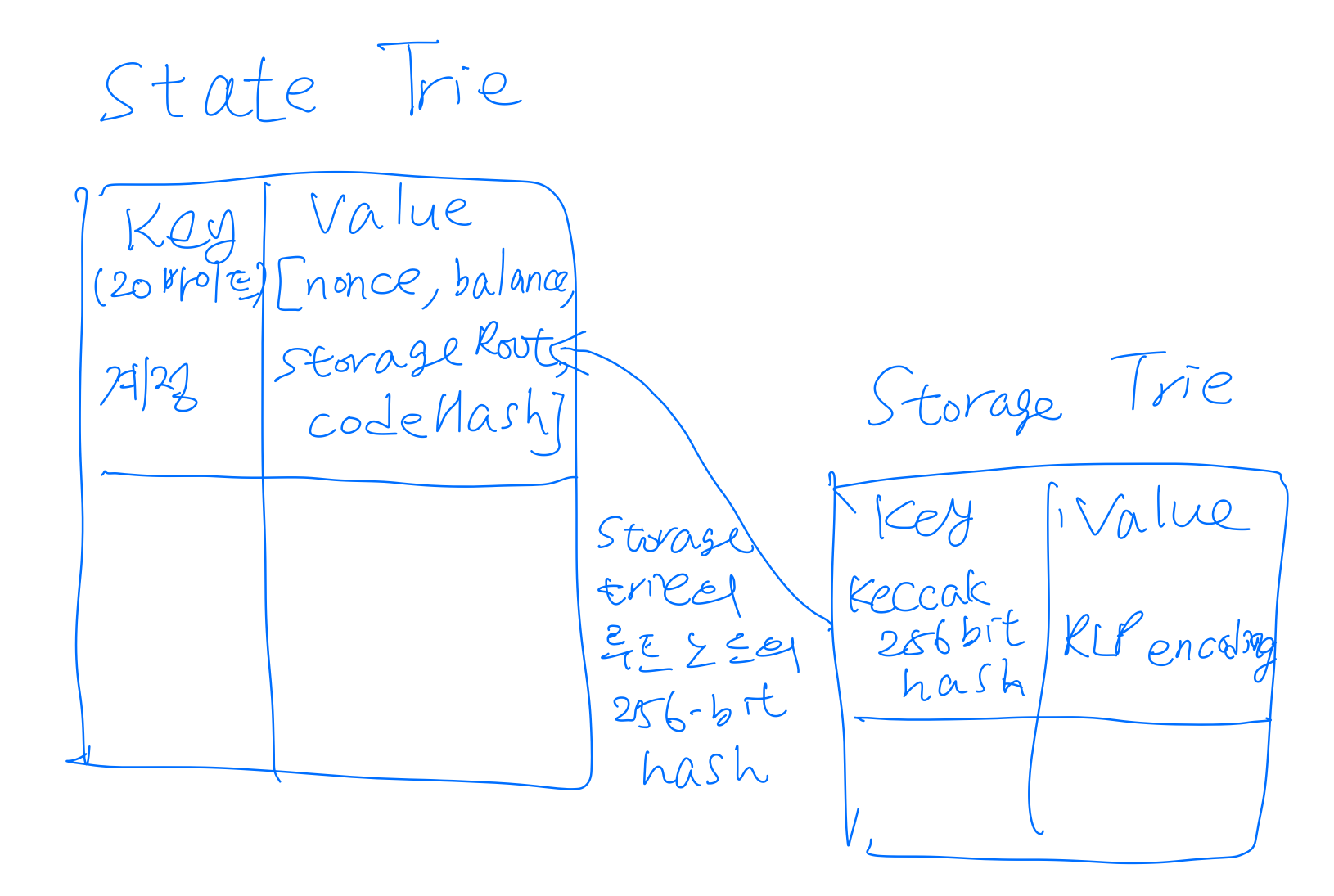
pjt\_dir> more eth\keystore\UTC--2019-01-03T21-57-03.966937024Z--21c704354d07f804bab01894e8b4eb4e0eba7451  
{"address":"21c704354d07f804bab01894e8b4eb4e0eba7451","crypto":{"cipher":"aes-12  
8-ctr","ciphertext":"e2d7e1e0d18de8f9e60d3e65ed54ffe29ad73e8eeb91e97396106c6ad1e  
993ae","cipherparams":{"iv":"3d8811a8de515619b255a309d5c47be9"},"kdf":"scrypt","  
kdfparams":{"dklen":32,"n":262144,"p":1,"r":8,"salt":"e99a544433dbd7764562adfb9a  
2bf3e176e49098b42c3d67325ec5d69ff18a75"},"mac":"45b06dc019cd4f7886c0360f2fb7b20e  
1d16ffca368c8b08366569de11da75a5"},"id":"64317eab-5d91-4add-abf8-a34303a01230","  
version":3}

### 2.2 계정의 상태정보

블록체인에는 매우 많은 계정이 존재하고 있고, 더욱 늘어나고 있다. 계정에 얼마나 잔고를 가지고 있는지 어떻게 추적할까 궁금할 것이다.

바로 State trie에서 한다. State trie는 모든 계정 각각의 현재 상태를 저장하고 있다. 따라서 매우 큰 저장 구조가 된다.

계정을 키로 사용하고 (그래서 키는 32바이트), 이러한 키별로 잔고, nonce, balance, storageRoot, codeHash 정보들을 저장한다. 이 가운데 stoargeRoot 항목은 storage trie의 root node를 가리키고 있다 (정확히 말하면 root hash를 가지고 있다).



alt text

### 계정 잔고

계정은 잔고(balance)를 가지고 있다. 통장의 비밀번호와 같은 역할을 하는 키를 사용해 소유를 증명하고, 입출금을 할 수 있다.

계정의 잔고는 어떻게 아는가? **비트코인**에서는 거래를 하면 입금-출금의 잔액이 남게 된다. 그 잔액은 거래송신자의 주소로 돌려주게 된다. 이를 UTXO(unspent transaction output)이라고 한다.

**거래의 기록에 그 잔고가 저장**된다. 비트코인은 모든 거래가 하나의 원장 (블록체인)에 기록된다. 통장이 있고, 그 계정의 거래만 그 통장에 기록되는 방식이 아니다.

잔고를 구하려면, **계정의 미사용 잔액의 합계**를 계산해야 한다. 거래기록이 발생하면서 잔고기록은 여러 기록에 분산되게 된다. 입금이 발생하면 그 잔고가 합쳐져서 기록되는 것이 아니라, 건별 기록이 남는다. 따라서 잔고는 집계되어 있는 것이 아니라, 여러 기록에 남겨진 **미사용잔액 UTXO**을 여기 저기 찾아서 합산해야 한다.

### nonce

개인계정에서 트랜잭션이 발생한 거래건수, **nonce** 정보를 가지고 있다. nonce는 0부터 시작한다.

계약 계정 역시 **nonce**를 가진다. 그러나 다른 컨트랙의 기능을 호출하는 경우에는 증가하지 않고, 다른 컨트랙을 생성하는 경우에만 증가한다.

**nonce**

**n**umber of **once**를 줄인 말로 보안에서는 딱 한 번만 쓰이는 수를 의미한다. 인증에 부여되는 식별자로서 이전의 인증정보를 사용하여 리플레이 공격을 방어하기 위해 사용한다. nonce는 2 종류가 있다. (1) **계정의 nonce**: 계정에서 전송된 트랜잭션 건수를 의미한다. (2) **작업인증의 nonce**: 작업인증 PoW를 하면서 Hash를 생성하기 위해 nonce를 선택한다. 이 때 선택되는 nonce를 의미한다.

eth.getTransactionCount() 함수는 현재 nonce 번호를 알려준다.

> eth.getTransactionCount(eth.accounts[0])  
52

nonce가 중요한 이유는 **이중거래**를 막을 수 있기 때문이다.

내 계정에서 트랜잭션 A, B, C를 전송하였다고 하자. 트랜잭션이 전송될 때마다 nonce도 하나씩 증가한다. 따라서 nonce가 적은 트랙잭션이 순서대로 먼저 처리되어야 한다. nonce 0을 처리하고 나면 2를 처리해서도 안되고 건너뜀 없이 1을 처리해야 한다.

굳이 이중거래를 시도한다면:

* 한 트랜잭션을 전송한다.
* 빠르게 트랜젹션의 결과를 취득한다.
* 첫 트랜잭션이 마이닝 되기 전, 빠르게 두번째 거래를 더 높은 gas비용으로 전송하여 두 번째 거래를 성공시키고 첫 째 거래를 무효화한다. 그러나 이러한 이중거래는 nonce를 거래마다 가지게 하여 처리순서를 결정하면 두 번째가 첫 째보다 먼저 처리될 수 없기 때문에 불가능하다.

### storageRoot

StorageRoot는 계정이 저장된 해시 값이다. storage trie는 컨트랙의 상태가 저장된다. 계정은 각각 자신의 storage trie를 가지고 있다. 그 storage trie 루트의 256 비트 해시 값을 storageRoot라고 한다.

### codeHash

CodeHash는 외부계정에게는 이 값이 없고, 내부 콘트랙은 코드의 해시 값이다.

## 문제 4-1: 계정을 생성하기

계정은 생성이 되고 나면 삭제할 수 없다. keystore 디렉토리에서 제거하면 삭제하는 효과를 얻을 수 있다.

명령창에서 직접 계정을 생성할 수 있다. 이 때 설정했던 --datadir을 올바르게 넣어야 한다.

pjt\_dir> geth --datadir ./eth account new

또는 콘솔창에서 newAccount() 명령어를 사용할 수 있다.

> personal.newAccount("password")

그리고 그 결과를 출력하면 새로 만들어진 계정을 포함하여 계정 목록을 출력한다.

> eth.accounts  
["0x21c704354d07f804bab01894e8b4eb4e0eba7451", ...생략..., "0x778ea91cb0d0879c22ca20c5aea6fbf8cbeed480"]

## 문제 4-2: 계정을 coinbase로 정하기

은행의 계정과 같이, 누구나 필요에 따라 여러 계정을 소유할 수 있다. 다만 이 가운데 급여를 받거나, 세금을 납부하는 계정이 있기 마련이다.

지급 또는 마이닝 보상을 받는 계정을 **coinbase** 또는 **etherbase**라고 한다. 여러 계정을 발급하면 순서가 주어지고, 첫째 계정이 coinbase가 된다.

accoun 0 대신 1로 coinbase로 변경하고 확인해보자.

> eth.coinbase  
"0xc8ea4c4e655f8152adc075a649aa7ec35564c7c0"  
> miner.setEtherbase(eth.accounts[1])  
true  
> eth.coinbase  
"0x4fa2c7caac80a8518264d263bdb5ed74f1a6f398"  
>

## 2.4 해싱

해시는 어떤 데이터를 입력하면 그로부터 생성해낸 **일정한 길이의 고유 값**이다.

입력 데이터의 **길이가 달라져도** 결과는 **일정한 크기**, 예를 들면 256비트의 문자열로 생성된다.

이 경우 의 가능성이 거의 없는 맞출 확률이 존재한다.

해시의 특징은:

1. 해시는 **원본을 되살릴 수 없다**. 해시는 원본에서 알고리즘에 따라 원본의 길이와 무관하게 고정 길이로 생성해낸 값이라 원본을 되살리기 극히 어렵다. (역상 저항성, preimage resistance). 즉 메시지 m에서 h를 구했으면 h = hash(m), h를 안다고 m을 알아내지 못해야 한다.
2. 그러나 해시는 **입력 데이터가 동일하면** 즉 수정되지 않는 한 항상 **같은 고유 값**을 생성해 낸다 (deterministic)
3. 원본이 **아주 조금 변경되어도 해시 값은 완전히 다른 고유 값**이 생성된다.
4. 입력 데이터가 다르면 해시 값이 다르다. 다른 운영체제, 다른 하드웨어에서 해싱한다해도 우연히라도 같은 경우가 있을 수 없다 (충돌 저항성 collision resistance).서로 다른 메시지 m1, m2의 해시가 hash(m1) = hash(m2) 서로 같지 않도록 한다

요약하면 해시 값에서 원본을 알아낼 수는 없지만, 해시 값이 동일하면 원본이 동일하다. 즉 송신자가 문서를 전송할 경우 해시 값을 만들어 같이 보내면, 수신자측에서 그 문서의 **위변조를 판별**할 수 있다. 수신자측에서 문서의 해시 값을 계산한 Hash\_Receiver와 받은 해시 값 Hash\_Sender과 비교 (Hash Validation) 하여 동일한 경우 원본과 동일하다고 판단할 수 있다.

많이 사용되는 해시 함수로는 MD, SHA가 있다. **이더리움은 Keccak hashing**을 사용하고, 반면에 **비트코인은 SHA-256**을 사용한다.

* MD2, MD4, MD5는 128비트 길이 해시값을 생성한다.
* SHA (Secure Hash Algorithm)
  + SHA-1은 160비트 길이
  + SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512 수는 비트길이)
  + SHA-3 가장 최근 2015년 NIST가 발표한 방식, **Keccak**이라고 부른다.

이더리움에서 해시 값의 활용:

1. 공개키를 해싱해서 그 해시 값으로 **주소를 생성**할 때 사용한다.
2. **무결성 검증**에 사용한다. **거래가 원본**임을 확인할 때 디지털 서명된 해시값과 비교하여 변조를 확인한다. 블록헤더의 해시 값으로 블록검증에 사용한다. **머클루트의 해시 값**을 계산하여 블록들의 변조가 있는지 확인한다.
3. 채굴하는 경우 해시 값을 계산하여 Target hash를 맞추는 경우 사용한다.

hash를 Python haslib라이브러리를 사용해서 이해해 보자. 우선 **md5 객체를 문자열을 넣어 생성**하자. 문자열은 길거나, 짧거나 상관없고, 유니코드로 인코딩해서 사용한다.

>>> import hashlib  
>>> md = hashlib.md5("Hello Hash".encode('utf-8'))

hexdigest()는 입력문자열의 hash결과를 16진수로 돌려준다. hash는 32 니블(nibbles) 즉 16바이트이다.

>>> print("hex hash: ", md.hexdigest())  
hex hash: 3d43d3cdf3e69cbe17cbfadc130ae2ed

digest\_size는 바이트 수를 나타낸다. md는 앞서 설명한 대로 128비트, 즉 16바이트이다.

>>> print("number of bytes: ", md.digest\_size)  
number of bytes: 16

이번에는 sha256으로 해시를 생성해보자. sha256은 32바이트, 즉 16진수로 64 니블이 생성된다.

>>> import hashlib  
>>> sha=hashlib.sha256("Hello Hash".encode('utf-8'))  
>>> print("hex hash: ", sha.hexdigest())  
hex hash: 6529ead5a42d94dcf8416b9192a6ae25c1700c006b4ef71ea7b4a67b34532996  
>>> print("number of bytes: ", sha.digest\_size)  
number of bytes: 32

아래 입력문자를 생성한 후 그 hash 값을 구해서 출력하고 있다. 입력문자는 nonce를 임의로 만들어 사용하는데 나중에 마이닝을 할 때 이런 방식으로 해시를 생성하게 된다. 그 nonce를 1부터 20까지 증가시키면서 문자를 생성하고 그 해시를 출력하고 있다. 그 결과를 보면:

* 모두 고정 길이로 출력되고,
* 입력 값이 조금만 달라져도 해시 값은 완전히 다르다는 것을 알 수 있다. 마이닝할 경우 nonce로 재해싱할 경우 해시 값이 이와 같이 달라진다.

>>> text = "Hello Hash"  
>>> for nonce in range(10):  
... # add the nonce to the end of the text  
... input = text + str(nonce) # adding nonce procudes totally different hashes.  
... hash = hashlib.sha256(input.encode('utf-8')).hexdigest()  
... print(input, '=>', hash)  
...  
Hello Hash0 => 34cc7b4455645fe634cd1ad246f2ae283bbade92d395cabf61608173764f0298  
Hello Hash1 => 20401add4f7fc728525f94c4b0e6be9516366aa4dd5766b569228783ddf526d0  
Hello Hash2 => 305b90d9b0bde189a842b9b55c9e6cba9c14f8188639544c4feea803b654cab9  
Hello Hash3 => 646b759044fcd4b12eea7c7196bedf6701e83b27380c887d2d7fe91a661ba422  
Hello Hash4 => 43025a967e5c0e0c1c92adb9361c970fa02ba889b37984f64179d057acef9d3d  
Hello Hash5 => 1f03762cfdc85f8a1562d981c893e4f2b2517ef4e221f6f12c91e8f69cb43cfb  
Hello Hash6 => e94dbbbb1dc0d918ab913de9509e1b47df2838fbbbf86fc2fcc9d8cd07535f8a  
Hello Hash7 => 85948326329e468cfa25153c54e90f3d11ebbbc7775f8716ad256478b09a6747  
Hello Hash8 => b7db3b7950c5af6eada985e7b1f88db39f3fbe25c95dc5fd3b96e6e9731b5446  
Hello Hash9 => 4c5f5b60e2e7d3b6629f05fdfe691777cf1ac99b1793d9aa589cc0904b69610b

이번에는 geth를 사용해서 해싱을 해보자. web3 라이브러리의 web3.sha()함수를 사용하여 문자열 해시를 생성할 수 있다. 위와 비교하면 "Hello Hash0" 해시 값이 서로 다른데, 위는 sha2 여기서는 sha3 Keccak을 사용하기 때문에 그렇다.

> web3.sha3(web3.toHex('Hello Hash0'))  
"0xb51061f1acf380094695cc6dd7a41628cb0de0cf8e7b24b0eec010c08863708b  
> web3.toHex('1')  
"0x1"  
> web3.sha3(web3.toHex('1'))  
"0x8a07523229fdc48491a5e56c76620ba40639eb940e6a2fbdf62b2799b4c86643"

물론 > 프롬프트에서 하지 않고 바로 명령창에서 해도 된다. Attach 명령창이 실행되고 있지 않아도 된다는 장점이 있다.

pjt\_dir> geth --exec "web3.sha3(web3.toHex('Hello Hash0'))" attach http://localhost:8445

## 2.5 디지털 서명

디지털 서명은 메시지의 전송자를 인증하기 위해 사용된다. 디지털 서명은 원본을 변경하는 것은 아니라, 원본을 기반으로 생성을 해서 서명이 만들어진다.

송신측에서는: - 메시지로부터 해시를 생성한다. - 그 해시에 개인키를 적용해서 디지털 서명을 생성한다.

수신측에서는: - 송신자가 해 놓은 디지털 서명을 공개키로 푼다. 당연히 풀리면 송신자가 맞다고 인증하게 된다. - 메시지의 해시가 서로 일치하면 원본과 일치한다고 인증하게 된다.

예를 들어 갑이 을에게 메시지를 전송한다고 하자. 갑은 자신의 키로 메시지에 디지털 서명을 하고 전송한다. 물론 메시지는 서명하기 전에 해싱해 놓아야 한다. 을은 갑의 공개키를 이용하여 메시지를 열람한다.

이 과정을 통해:

* 원본은 해싱하고 디지털 서명을 해 놓았기 때문에 원본이 **위변조 되지 않았음을 증명**할 수 있다. **어떤 악의적인 사용자가 원본을 수정하려고 한다고해도, 해시 값이 달라지기 때문에 그런 시도는 불가능**하다.
* 원본은 키로 풀어야만 열람할 수 있게 **송신자를 증명**할 수 있다. **수정했다고 해도 원본의 서명에 사용했던 개인키를 알 수 없으니 서명 또한 가능하지 못하다**.

더 알아보기: 비대칭키와 대칭키

대칭키는 전송측에서 암호화에 쓰이는 키와 수신측의 복호화에 쓰이는 키가 동일한 경우, 즉 암호키와 복호키가 동일하다. AES(Advanced Encryption Standard)를 예로 들 수 있다. 반면에 비대칭키는 암호키와 복호키가 서로 다르다. RSA 또는 ECDSA가 해당된다.

개인키(private key)는 디지털 서명에 사용한다. **개인키로 서명**하고, 받는 측에서 공개키(public key)로 검증한다.

반대로 **공개키로 사인**할 수 있다. A가 B에게 전송할 경우, A가 상대방 B의 공개키를 적용하고, B는 받아서 자신의 개인키를 이용하여 원데이터로 복원할 수 있다.

앞서 설치한 bitcoin 라이브러리를 사용해서 사인, 검증을 해보자. 앞서도 생성하였지만, 우선 사인에 필요한 키를 생성해보자.

>>> import bitcoin  
>>> privKey = bitcoin.sha256('hello key')  
>>> pubKey = bitcoin.privtopub(privKey)  
>>> addr = bitcoin.pubtoaddr(pubKey)  
>>> print("privKey: ", privKey)  
privKey: 3e295c8dc78fb7f3865067dfc42fe845263db7661296e7e32e3a37baa1a27a7b  
>>> print("pubKey: ", pubKey)  
pubKey: 04f6cc26cec156c180f8a215cf54db7799d0d42179f1e0b628cf364f09da95f17d5aab7edeeb1f529137a241d07cec555b2d8ec44a4cd24e87cf98001d428f564f  
>>> print("addr: ", addr)  
addr: 1NthZ9kJVbtxrHQiocfjLLTcMH3F2DLcgD

송신측에서 약속을 정하는 메시지를개인키로 사인을 해서 보낸다.

>>> msg="let's meet in my office at 10 in the morning."  
>>> sig=bitcoin.ecdsa\_sign(msg,privKey)  
>>> print("signature: ",sig)  
signature: Gxg7F5VjSyguXWt0TGKt5op0iQZVDK22Pa1aumPpC5gRbhc1iAPdAK4p+MZNRbQcFz8BTvljoT9YYv88pTZvw30=

수신측에서는 공개키로 그 메시지를 검증해서 원본임을 확인하게 된다.

>>> print("verified: ",bitcoin.ecdsa\_verify(msg,sig,pubKey))  
verified: True

누군가 10시 약속을 위조하여 12시로 변경한 후 공개키로 검증해 보자.

>>> msg1="let's meet in my office at 12 in the morning."  
>>> print("verified: ",bitcoin.ecdsa\_verify(msg1,sig,pubKey)) # wrong msg  
verified: False

공개키는 말 그대로 누구나 알 수 있는 키이다. 개인키로 서명된 메시지에서 공개키를 구할 수 있다. ecdsa\_recover(msg,sig) 함수를 관찰하면 메시지와 서명된 메시지를 인자로 받고 있다. 우리가 위에서 구한 공개키와 비교해 보면 동일하다는 것을 알 수 있다.

>>> print("pub key", bitcoin.ecdsa\_recover(msg,sig))  
pub key 04f6cc26cec156c180f8a215cf54db7799d0d42179f1e0b628cf364f09da95f17d5aab7edeeb1f529137a241d07cec555b2d8ec44a4cd24e87cf98001d428f564f

## 2.6 ECDSA

이더리움의 키는 ECDSA 알고리즘을 적용해서 만든다. 이 알고리즘은 해킹할 수 없을 정도로 견고하고, 그래서 블록체인은 매우 튼튼하고 해킹의 위험이 없다고 봐도 좋다.

ECDSA 함수는 로 표현되는 타원곡선 elliptic curve이다. 여기서 a, b를 변형해서 함수를 정한다.

아래 그래프에서 보듯이 타원곡선의 특징은:

* **x축으로 대칭**이고,
* 타원곡선에 **2점 P,Q을 지나가는 직선을 그으면, 반드시 3번째 R도 교차**한다.

함수의 그래프를 코드를 작성해서 그려볼 수 있다. 그러기 위해서 다음을 실행하려면 numpy, matplotlib을 설치해야 한다. 하나씩 설치하지 않고 한꺼번에 pip install numpy matplotlib 하면 된다.

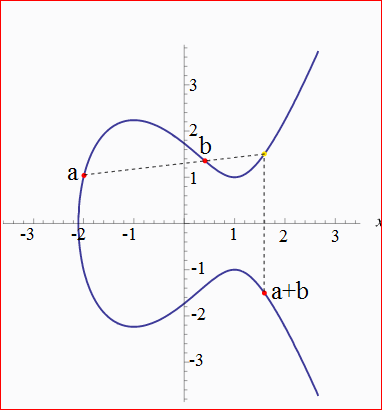
>>> import numpy as np  
>>> import matplotlib.pylab as pl  
  
>>> def f(x):  
... return x\*\*3 -3\*x + 4  
  
>>> xa = np.mgrid[-10:10:50j] # -10 ~ 10사이의 복소수 50개의 배열을 생성  
>>> y = list()  
>>> x = list()  
>>> for i in range(0,len(xa)):  
... \_x = xa.ravel()[i] # 2차원을 1차원배열로 변환후 순서대로 선택  
... \_y=f(\_x)  
... if \_y> 0:  
... y.append(np.sqrt(\_y))  
... x.append(\_x)  
  
>>> pl.plot(x, y, "-")  
>>> pl.plot(x, [-yi for yi in y], "-")  
>>> pl.show()

이 프로그램을 실행하면, 아래 호리병을 뉘운 모양의 ECDSA 곡선이 그려진다.

### Elliptic curve 연산

공개키는 타원곡선에 속하는 점의 곱셈 연산으로 구하게 된다.

* **더하기** 연산 (아래 그림 참조)
  + Elliptic Curve에 2 점을 선택해 직선을 그으면 한 점과 반드시 교차하게 된다.
  + x축에 수직으로 내려서 곡선과 만나는 점을 구하면 덧셈이 된다. 즉 **R의 대칭점**이다.
* **곱하기** 연산
  + 자신의 접점에서 더하기 과정을 하면 곱셈이 된다.
  + 접점은 당연히 P,Q가 동일해서 한 점이 되는 경우이고, 이 점에서 **접선(tangent line)**을 그려서 만난 **R의 대칭점**이다. 그 결과는 P+P=2P, 즉 2를 곱해서 얻은 값이다.



alt text

### 더하기 연산으로 구하는 공개키

시작점 'Generator' G에 정수k를 계속 더하면 또는 2를 곱해 나가면, 그 결과 값인 P는 커브에 존재하는 점으로 계산된다.

P=k\*G

이 때 **k가 개인키**, **P가 공개키**가 된다. 즉 pubKey는 privKey에서 생성되며, 타원곡선 상에 존재하는 **x,y 좌표 값**이다.

privKey는 256비트 **1 ~ 에 존재하는 하나의 값**이므로, 그 사이의 수만큼 계속 더해야 또는 곱해 나가야 공개키가 된다.

pubKey = privKey \* G

그렇다면 **pubKey는 공개키이고 이로부터 비밀번호에 해당하는 privKey를 알아낼 수** 있을까?

여기서 PubKey=privKey \* G는 계산이 상대적으로 매우 빠르고 쉽지만, 역으로 pubKey/G=privKey 즉 **pubKey에서 privKey를 계산해 내기는 거의 불가능하다**. G를 계속 privKey 횟수 만큼 빼주어야 찾을 수 있다. 적어도 **privKey의 평균 값인**  횟수를 시도해야 한다.

아무리 성능이 좋은 컴퓨터라더라도 그 횟수만큼 연산해서 privKey 암호를 해킹하는 것은 매우 어렵다는 의미이다.

### secp256k1

secp256k1은 비트코인, 이더리움 등 여러 암호화폐에서 사용하는 공개키를 구현하기 위해 사용하는 타원곡선 Elliptic curve의 이름을 말한다.

그러니까, 타원곡선 함수를 딱 특정해서 정하고, 그 함수를 지칭해서 secp256k1이라고 이름을 붙여 놓은 것이다. 명칭이 상당히 난해하겠지만, 앞 글자 sec는 효율적 암호화를 위한 표준 (Standards for Efficient Cryptography)을 의미하니까 그 목적을 분명히 말하고 있다.

함수를 특정하다니 왜 그럴까? 이 함수를 쓰지 않으면 무작위로 인자를 정해야 하지만, 여기서는 미리 정해 놓겠다는 것이다.

타원곡선 함수를 설정해 놓으면, 계산에 걸리는 **시간이 약 1/3이 줄어들어 효율적이기 때문에 필요한 인자들을 미리 설정**해 놓고, 그 정해진 인자를 사용하는 타원곡선함수를 secp256k1이라고 지칭하고 있다.

secp256k1에서는 필요한 6개 인자 p, a, b, G, n, h를 아래 표에서 보이는 값으로 정해놓는다. 표에서 보듯이, a = 0, b = 7로 미리 정해 놓고, 타원곡선 함수는 로 표현할 수 있다.

|  |  |
| --- | --- |
| 구분 | 값 |
| p (prime 소수) | FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F = 2256 - 232 - 29 - 28 - 27 - 26 - 24 - 1 |
| a (타원곡선 함수의 계수) | 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 |
| b (타원곡선 함수의 계수) | 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007 |
| G (시작점) | 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 |
| n (G를 자신에게 계속 더하면서 nG=O이 되는 n) | FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141 |
| h (cofactor) | 01 |

### 파이썬으로 유한체 Elliptic curve 이해하기

파이썬으로 Elliptic Curve를 생성해보자.

그러기 위해서는 라이브러리 Pycoin을 설치해야 한다. 동일한 명칭으로 cdecker/pycoin와 richardkiss/pycoin 2 개가 있으니, 주의해서 후자를 설치한다. **매뉴얼에 3.6, 3.7을 지원**한다고 했지만 최신 버전은 잘 설치가 되지 않는다. python2로 해보거나, python3에서는 pycoin 버전을 낮추어서 0.77로 해보자.

pip install pycoin==0.77 # 파이썬 3를 사용하면 pycoin의 특정 버전 사용. 윈도우에서는 pip대신 pip3 사용  
pip install python2-secrets # python2에서 사용하는 경우

pycoin이 잘 설치 되었으면, 명령창에서 ku (key utility, 키를 관리하는 명령어)를 입력하면 해당 키가 출력된다 (https://github.com/richardkiss/pycoin/blob/master/COMMAND-LINE-TOOLS.md). 참고로 ku는 BIP32 키, WIF, 비트코인 및 알트코인 주소를 지원한다.

C:\Users\jsl\Code\201711111>where ku 명령어가 설치되어있는지 먼저 확인하자 (Python 3.7기준 아래 디렉토리에 설치)  
C:\Program Files (x86)\Python37-32\Scripts\ku.exe  
  
C:\Users\jsl\Code\201711111>ku 1  
  
input : 1  
network : Bitcoin mainnet  
symbol : BTC  
secret exponent : 1  
 hex : 1  
wif : KwDiBf89QgGbjEhKnhXJuH7LrciVrZi3qYjgd9M7rFU73sVHnoWn  
 0C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D 64  
 uncompressed : 5HpHagT65TZzG1PH3CSu63k8DbpvD8s5ip4nEB3kEsreAnchuDf  
public pair x : 55066263022277343669578718895168534326250603453777594175500187360389116729240  
public pair y : 32670510020758816978083085130507043184471273380659243275938904335757337482424  
 x as hex : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798  
 y as hex : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8  
...생략...

**유한체 타원곡선(Elliptic curve over the finite field)**는 modulus 연산을 하여 범위를 넘어서는 경우를 제한하게 된다. **더하기 또는 곱하기를 하여도 그 결과는 항상 유한체에 존재**하게 된다. 따라서 x,y 좌표로 생성된 **pubKey는 그 크기가 일정 범위** 에 있게 된다.

상수 a=-3, b=4, modulus=29인 타원곡선을 식으로 표현하면:

먼저 타원곡선 함수를 정의해보자.

>>> import pycoin.ecdsa as ecdsa  
  
>>> e=ecdsa.ellipticcurve.CurveFp(29,-3,4)  
>>> print(e.p(), e.a(),e.b())  
 29 -3 4

특정 요소가 타원곡선에 존재하는지 확인할 수 있다. 시작점 (0,2)와 (22,1)은 곡선 상에 존재하지만, (22,3)은 그렇지 않다.

>>> print("should be True (on curve) -> ", e.contains\_point(0,2))  
should be True (on curve) -> True  
>>> print("should be True (on curve) -> ", e.contains\_point(22,1))  
should be True (on curve) -> True  
>>> print("Should be False (not on curve) -> ", e.contains\_point(22,3))  
Should be False (not on curve) -> False

G (0,2)는 시작점이고 곱셈으로 유한체 상의 요소를 생성해 보자. 그래프를 보면 타원곡선의 모양의 점들과 그래프 범위를 넘어선 경우 모듈러스 29 연산의 나머지를 표시한 점들을 볼 수 있다.

>>> import numpy as np  
>>> import matplotlib.pyplot as plt  
  
>>> G=ecdsa.Point(e,0,2)  
>>> print("The generator point", G)  
>>> for k in range(1,31):  
... print(k, str(k\*G))  
... P=k\*G  
... plt.scatter(P.x(),P.y())  
... plt.text(P.x(),P.y(),str(P))  
  
>>> plt.xlabel('x')  
>>> plt.ylabel('y')  
>>> plt.show()

위 코드 for반복문이 실행되면 다음과 같이 함수의 점들이 출력된다.

The generator point (0,2)  
1 (0,2)  
2 (6,17)  
3 (22,1)  
...생략...  
28 (22,28)  
29 (6,12)  
30 (0,27)

### 개인키 다시 생성해 보기

앞에서 개인키를 생성해 보았지만, 여기서도 ecdsa함수에서 생성된 무작위 수가 privKey가 된다. 64 니블, 즉 32바이트이다.

앞에서 bitcoin을 이용해서 만들었던 개인키와는 다른 값이 나온다. 여기서는 pycoin의 함수를 이용해서 하고 있고 secp256k1을 이용하고 있다. 즉 키 생성 알고리즘이 달라서 그런 것으로 이해하자.

Python secrets은 SystemRandom 클래스를 사용하여 무작위수를 생성하는 모듈 randbelow()는 괄호 이하의 무작위 수를 생성한다. 무작위 수에서 사적키가 생성되므로, 매 번 다를 수 있다.

>>> import secrets  
>>> import pycoin.ecdsa as ecdsa  
  
>>> privKey=secrets.randbelow(ecdsa.generator\_secp256k1.order())  
>>> print("private key: ", privKey) 사적키를 10진수로 출력한다.  
private key: 28935648531180725967467340336175804302110662174298870838936251657389179208495  
>>> print("private key in hex: ", hex(privKey)) 사적키를 16진수로 출력한다.  
private key in hex: 0x3ff8ff270f52543b50704b521aa339a17dc9539264446b3c8b7b930d6281032f

### 공개키 다시 생성해 보기

secp256k1 시작점과 privKey의 곱셈 연산에서 pubKey를 구한다.

>>> pubKey = (ecdsa.generator\_secp256k1 \* privKey).pair()  
>>> print("pubKey: ", pubKey)  
pubKey: (93887791040130844862898420058609825954578110834910496421245258186192394357504, 57149901476873663228518799976060047377824158452386513969303714190024656145636)

### 사인

hashlib.sha3\_256() 해싱함수로 해싱을 한다. int.from\_bytes()는 Python 명령어로 바이트로부터 정수를 생성한다.

그리고 sign(ecdsa.generator\_secp256k1,privKey,msgHash) 사적키로 메시지의 해시에 대해 secp256k1 사인을 한다.

>>> import hashlib  
>>> import codecs  
  
>>> msg="let's meet in my office at 10 in the morning."  
>>> hashBytes = hashlib.sha3\_256(msg.encode("utf8")).digest()  
>>> #msgHash=int(codecs.encode(hashBytes, 'hex'), 16) # python2 & 3  
>>> msgHash=int.from\_bytes(hashBytes, byteorder="big") # was working for python3  
>>> signature=ecdsa.sign(ecdsa.generator\_secp256k1,privKey,msgHash)  
>>> print("signature: ", signature)  
signature: (32490553339647232928920509187700038770541835740050421513455566426914005112153, 50999034356653861730528767813034385846900368366359316685311359225748505618748)

### 인증

verify(ecdsa.generator\_secp256k1,pubKey,msgHash,signature)는 공중키로 사인된 메시지를 인증한다. 여기서 보듯이 공적키로 인증을 한다.

>>> valid=ecdsa.verify(ecdsa.generator\_secp256k1,pubKey,msgHash,signature)  
>>> print("valid: ", valid)  
valid: True

### v, r, s

v, r, s는 서명할 때 넣어주고, 이 값을 이용하여 공개키를 알아낼 수 있다. 곧 5장에서 배우게 될 거래를 구성하는 필드이다.

v, r, s를 출력해 볼 수 있는 라이브러리로 eth\_keys가 있는데, 설치하려면 pip install eth\_keys하면 된다.

위 코드에서 사적키는 임의로 01의 32바이트로 설정하고 있다. 출력하면 예상한 결과를 볼 수 있다.

>>> from eth\_keys import keys  
>>> pk = keys.PrivateKey(b'\x01' \* 32)  
>>> print(pk)  
0x0101010101010101010101010101010101010101010101010101010101010101

공적키는 사적키에서 타원곡선 함수를 적용해서 구하고 있다. 코드에서 pk.public\_key로 출력할 수 있다.

>>> print(pk.public\_key)  
0x1b84c5567b126440995d3ed5aaba0565d71e1834604819ff9c17f5e9d5dd078f70beaf8f588b541507fed6a642c5ab42dfdf8120a7f639de5122d47a69a8e8d1

메시지를 서명하고 나서 그 결과를 출력하면, 65 바이트이다.

>>> signature = pk.sign\_msg(b'a message')  
>>> print(signature)  
0xccda990dba7864b79dc49158fea269338a1cf5747bc4c4bf1b96823e31a0997e7d1e65c06c5bf128b7109e1b4b9ba8d1305dc33f32f624695b2fa8e02c12c1e000   
>>> print(len(signature))  
65

그리고 디지털 서명으로부터 v, r, s 값을 구할 수 있다.

>>> print("v: ", signature.v) # v는 1바이트로 0,1,2의 값을 가질 수 있다 (또는 27,28).  
v: 0  
>>> print("r: ", signature.r) # 디지털 서명 64바이트에서 출력할 수 있다.  
r: 92658050108416009874678855014183140825758716366670817325881885506266554472830  
>>> print("s: ", signature.s) # 디지털 서명 64바이트에서 출력할 수 있다.  
s: 56592813748953986272943098462069080518121627637438283188421834314135744201184

## 문제 4-3: 마이닝 해보기

geth를 시작할 때부터 마이닝 옵션을 설정할 수 있다. --mine은 마이닝을 실행하겠다는 옵션이고 --minerthreads는 마이닝에 몇 개의 CPU 쓰레드를 사용할 것인지 (기본은 0) --datadir는 현재 geth가 실행되고 있는 디렉토리이다.

pjt\_dir> geth --datadir "./eth" --mine --minerthreads=4 --port 30445

또는 geth console에서 마이닝을 할 수 있다. 공중망이나 테스트망에서는 마이닝을 굳이 별도로 실행할 필요는 없다. 사적망에서는 자신이 거래를 발생시키고, 그 한 건만을 마이닝할 수 있다.

1 건만 마이닝할 경우는 그 숫자를 sleepBlocks(1) 함수의 인자로 넘겨주면 된다. miner.start(1) 함수의 숫자는 할당 쓰레드의 숫자이다.

> miner.start(1);admin.sleepBlocks(1);miner.stop()  
null

## 2.6 Wallet

Wallet은 우리가 일상생활에서 사용하는 현금, 신용카드 등을 가지고 있는 지갑과 비슷하다. Wallet은 **계정, 개인키**를 가지고 있는 파일 또는 데이터베이스를 말한다.

단순히 계정만을 가지고 있지 않고 **블록체인과 인터페이스**하는 기능, 잔고를 조회하거나 스마트계약을 블록체인에 배포하는 등도 가능하다.

지갑은 주소를 생성하는 기능도 있는데, 매번 완전히 새롭게 생성하지 않고 **HD(Hierarchical Deterministic)** 방식으로 하나의 마스터 시드 Seed 키에서 다수의 주소를 생성할 수 있다. 이런 암호화페 지갑을 계층적 결정 지갑이라고 한다.

Wallet은 형식에 따라 PC Wallets, 모바일 Wallets, 하드웨어 wallets, USB wallets, 종이형태의 wallets이 있다. QR코드로 만들어진 Wallet도 있다.

소프트웨어 형식으로 존재하는 MetaMask, Mist (2019년 3월 개발 중단), MyCrypto (https://mycrypto.com) 등을 사용할 수도 있다.

geth도 지갑기능을 제공한다. 계정을 만들고, 입출금, 전송을 할 수 있다. wallet은 네트워크에 저장되어 있지 않고, 사용자 컴퓨터 /keystore에 있다. 아래에서 보듯이 형식은 UTC--{year}-{month}--{account} 출력된다.

pjt\_dir> dir eth\keystore (pjt\_dir\eth\keystore 폴더에 설치된 경우)  
03/22/2020 05:42 PM 491 UTC--2020-03-22T08-42-34.444935200Z--c8ea4...64c7c0  
03/22/2020 05:46 PM 491 UTC--2020-03-22T08-46-05.181606000Z--4fa2c...6f398  
03/28/2020 10:44 AM 491 UTC--2020-03-28T01-44-04.678799900Z--ce5cd...166a6

또는 geth 단말에서 personal.listWallets 명령어로 지갑의 계정 내용과 현재 잠금 상태를 출력할 수 있다.

geth> personal.listWallets  
[{  
 accounts: [{  
 address: "0x21c70...생략...a7451",  
 url: "keystore:///...생략.../eth/keystore/UTC--2019-01-03T21-...생략...a7451"  
 }],  
 status: "Unlocked",  
 url: "keystore:///...생략.../eth/keystore/UTC--2019-01-03T21-...생략...a7451"  
}, {  
 accounts: [{  
 address: "0x778ea...생략...ed480",  
 url: "keystore:///...생략.../eth/keystore/UTC--2019-01-04T01-...생략...ed480"  
 }],  
 status: "Locked",  
 url: "keystore:///...생략.../eth/keystore/UTC--2019-01-04T01-...생략...ed480"  
}]

## 연습문제

1. 자신의 컴퓨터에서 계정은 하나가 아니라 복수를 생성할 수 있다. 계정을 정렬해서 순서가 결정된다. OX로 답하시오.
2. 잔고확인 함수 eth.getBalance()는 어떤 인자를 넘겨주어야 하는지, 계정0의 잔고를 계산하는 명령문을 적으시오.
3. coinbase는 몇 번째 계정을 말하는지 답하시오.
4. coinbase를 계정1로 변경하는 명령문을 작성하시오.
5. nodeInfo에서 확인할 수 있는 정보가 아닌 보기를 고르시오.
   1. enode 2) ip주소 3) difficulty 4) ChainId 5) nonce
6. enode에 포함되는 정보가 아닌 보기를 고르시오.
   1. 공개키 2) ip 3) 포트번호 4) ChainId
7. 네트워크의 peer를 찾기 위해 무작위로 메시지를 보낸다. 못찾으면 peer를 수동으로 추가할 수 없다. OX로 답하시오.
8. hash rate의 계산식을 적으시오. 계산식을 설명하시오.
9. 은행에서는 개인 또는 법인이 계정을 가질 수 있다. 블록체인에서는 컨트랙도 계정을 가질 수 있다. OX로 답하시오.
10. 블록체인의 스마트 컨트랙에서 개인계정으로 송금할 수 있다. OX로 답하시오.
11. 개인키는 몇 바이트인지 적으시오.
12. 공개키는 몇 바이트인지 적으시오.
13. 주소(address)는 몇 바이트인지 적으시오.
14. 공개키를 생성하는 알고리즘 명칭을 적으시오.
15. 우리가 보는 개인키는 이미 WIF형식으로 만들어져 있다. 그 개인키를 원시 그대로 볼 수 없다. OX로 답하시오.
16. nonce는 리플레이 공격을 방어하기 위해 사용한다. OX로 답하시오.
17. 이더리움에서 해싱하는 방식을 적으시오.
18. 해싱은 입력이 동일하면, 항상 동일한 값을 생성한다. OX로 답하시오.
19. 해싱할 때 nonce값을 더하면, 해싱 문자열은 더한 부분만 수정된다. OX로 답하시오.
20. 이더리움의 디지털 서명은 공적키로 서명하고 사적키는 필요하지 않다. OX로 답하시오.
21. 이더리움은 익명이라서 지갑에는 사적키는 없고 잔고만 보유하고 있다. OX로 답하시오.
22. 지갑은 컴퓨터에만 있을 수 있다. OX로 답하시오.
23. 주소를 생성할 때, 매번 seed를 가지고 생성할 수 있지만, 그렇지 않고 하나의 마스터 시드에서 생성할 수 없다. OX로 답하시오.
24. 이더리움에서는 비대칭키로 암호화를 한다. OX로 답하시오.
25. 다음 항목을 실행하는 프로그램을 작성하시오. 문제를 풀기 전, 계정을 2개 더 생성하고, 잔액을 5 ether 이상이 되도록 충전한다. 파일은 exercise1.js로 저장하고 명령창에서 geth --exec로 실행하시오. console.log("Block Number:" , eth.blockNumber) 이런 방식으로 메시지도 출력되도록 작성하시오.
    1. 현재 블록번호를 출력
    2. 자신의 enode 값을 출력
    3. peer가 있는지? 있으면 그 peer를 출력
    4. 계정 목록을 출력
    5. 각 계정의 잔액을 ether로 출력
    6. 코인베이스를 2번째로 변경하고, 변경전과 변경후의 coinbase를 출력
    7. 현재 대기하는 트랙잭션 수를 출력. 없다면 그 이유를 메시지로 출력
    8. 현재 블록번호를 출력. 블록번호가 처음 출력했을 때와 변동이 있는지? 계정을 출력하거나, 잔고를 출력하는 등 몇 차례 작업이 있었는데도 불구하고 블록 번호가 그대로인 이유를 메시지로 출력.
26. 내용이 위조되면 해싱의 출력이 얼마나 변경되는지 확인하시오. 1907년 고종의 a plea for korea로 알려진 "헤이그 만국 평화 회의에 고하는 글"의 전문 중 "일본인들은 대한제국 황제 폐하의 정식 허가 없이 행동하였다."라는 문장이 있다. 영문으로는 "The Japanese acted without the consent of his Majesty the Emperor of Korea."이다. 이 문장의 without을 with로 위조했다고 하자. 즉:

* (원본) "The Japanese acted without the consent of his Majesty the Emperor of Korea."
* (위조) "The Japanese acted with the consent of his Majesty the Emperor of Korea."

파이썬 프로그램으로:

* 1. 원본을 해싱하고 출력
  2. 위조를 해싱하고 출력
  3. 원본은 위조보다 영어 3 글자가 많다. 원본과 위조에서 구한 해시의 바이트 수와 차이를 출력

1. 메시지를 인증해서 위조가 있는지 확인하시오. 위 고종의 헤이그 만국 평화 회의에 고하는 글은 끝내 전해지지 못했다. 현재의 블록체인 기술을 사용하였다면 원문을 무사히 전달했을 수 있어서 아쉽다. 파이썬 프로그램으로:
   1. 위 문장1)을 사인해서 인증하시오. 사인과 인증 결과를 출력
   2. 문장2)를 사인하고, 1)과 다른지 비교하고, 인증 결과를 출력
2. 다음을 프로그램해서 geth --exec loadScript로 실행하시오. 마이닝 한 건에 얼마나 충전이 되었는지 확인하시오.

* 코인베이스 출력
* 코인베이스의 잔고 출력
* 마이닝 1건 실행
* 잔고 증가분을 계산하여 출력