이더리움 프로그래밍 수업(1)

2017년 7월 13일 목요일 오후 9:32

박재현

jaehyunpark.kr@gmail.com

http://wisefree.tistrory.com

들어가며

비트코인과 블록체인, 그리고 이더리움에 대한 여러 자료들을 관련 사이트 등에서 찾아보면서 해당 내용이 이더리움 플랫폼의 최신 내용들과 다른 내용이 많아 테스트를 하거나 실제 프로그래밍을 하는 데 소소한 어려움이 많은 것을 느꼈습니다. 그리고 Ethereum.org 공식 채널을 포함, 주변에서 검색을 통해 접할 수 있는 이더리움 개발 관련 자료들이 적고 , 오래된 자료들이 많은 상황입니다. 따라서 Seoul Ethereum Meetup 멤버들과 기타 이더리움 개발을 하려는 분들의 손쉬운 이더리움 입문을 위해 포스팅을 시작합니다. 저도 함께 배우는 과정이기에 다소 부족한 부분이 많을 수 있으나 함께 채워나갔으면 합니다.

Mac 운영체제를 기준으로 작성하며 , 만약 이 포스팅에 문제가 발생

할 경우 알려주면 이를 바로 잡을 계획입니다.

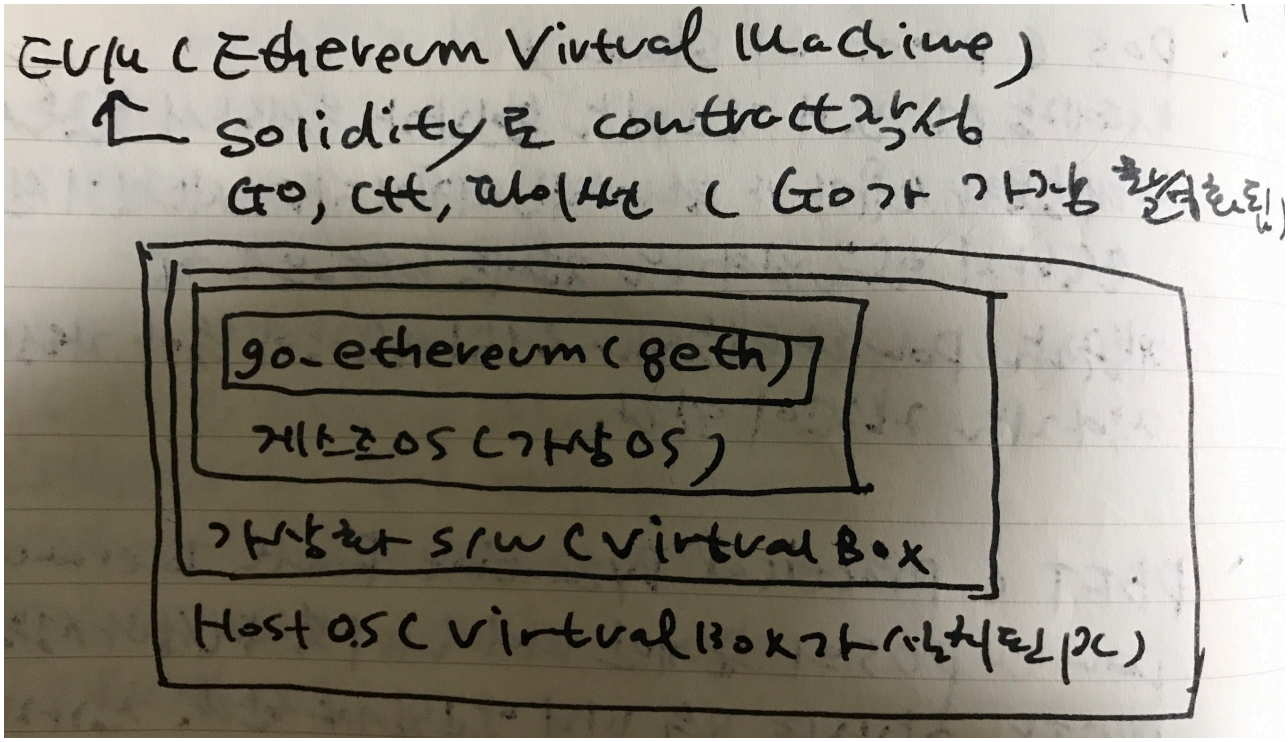
이번 가이드를 통해 이더리움의 구동 환경을 이해하고 , 실행환경을 구축하며 , 이더리움의 가상화폐인 Ether를 발행하고 이를 다른 사용자에게 송금하면서 이더리움의 기본인 가상화폐를 다루는 것을 익힐 수 있습니다.

이더리움 프로그램 구동 환경 이해

다음은 간략한 이더리움 실행 환경입니다. Virtual Box 라는 가상화 엔진상에 geth라는 EVM(Ethereum Virtual Machine)가 위치합니다. 이 코어 엔진상에서 실행가능한 프로그램을 개발하는 것이 이더리움 프로그래밍입니다.

자바스크립트나 파이썬 같은 기존의 언어들을 사용하여 프로그램을 작성해도 EVM에서 구동이 되기 때문에 이더리움을 튜링 컴플리트(Turing Complete)라고 합니다. 어렵게 생각말고 지금은 이더리움에서 일반 개발 언어를 사용해서 가능한 모든 것을 작성할 수 있다" 라고 이해하면 됩니다. 참고로 , 비트코인은 튜링 컴플리트가 아닙니다. 보안 등 여러 이유로 if 문 만을 제공하는 등 의도적으로 제한된 범위 내에서만 개발을 지원합니다.

현재 이더리움 엔진은 Go 언어와 C++ , 파이썬 등으로 개발되었고 Go언어 만든 Go-Ethereum이 가장 업데이트가 활발합니다(바이너리 이름이 geth입니다). 본 글에서도 Go-Ethereum을 사용합니다. 다음은 앞서 설명한 내용을 정리하였습니다. 본래 노트에 글쓰기를 좋아해서 노트한 것을 그냥 올립니다. 지저분해도 이해해 주세요.

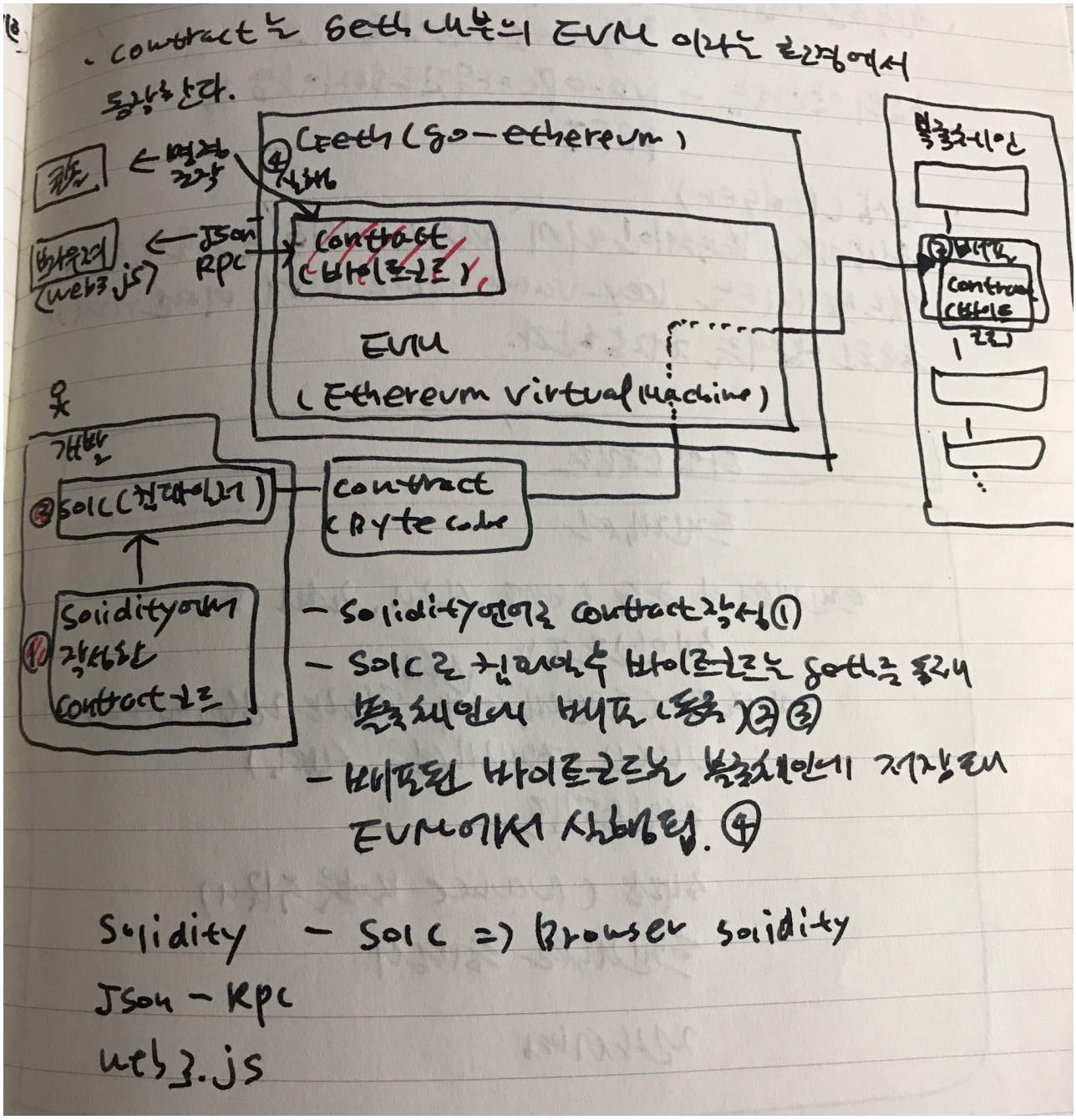


이더리움 엔진인 geth 은 3가지 인터페이스를 통해 활용이 가능합니다 -- (1) HTTP JSON RPC (2) web3.js 를 통한 자바스크립트 언어 (3) Solidity .

위의 인터페이스를 다양한 언어로 이더리움 클라이언트를 개발할 수 있습니다. 가령, 자바 스크립트로 Web3.js를 사용하여 이더리움 클라이언트를 개발할 수 있습니다. 또는 JSON RPC 호출 후 자바로 클라이언트를 개발할 수도 있습니다. 그러나 이더리움 개발의 꽃인 Smart Contract를 개발하기 위해서는 Solidity를, Serpent, LLL 언어를 사용해야 합니다. 이 중 가장 각광받고 있는 Solidity 를 사용하겠습니다. 참고로 , 아직 Solidity는 개발 툴의 기능도 부족하고 여러면에서 부족하나 이더리움의 성장과 더불어 개발자에게 스스로의 가치를 높일 수 있는 좋은 기회가 될 것 입니다.

다음은 이더리움의 전체 구동 환경을 정리한 그림입니다. 먼저 상단의 왼쪽을 보면 콘솔과 브라우져가 등장합니다. 사용자는 geth를 구동시킨 후 콘솔을 통해 일련의 콘솔 명령어를 통해 원하는 기능을 geth에게 지시할 수 있습

니다. 더불어 Json RPC와 Web3.js 자바스크립 라이브러리로 작성된 프로그램을 브라우져를 통해 구동시킬 수 있습니다.



하단부 왼쪽은 Smart Contract에 대한 구동 환경입니다. 개발자는 Solidity로 프로그램을 작성한 후 Solc 컴파일러를 통해 컴파일을 합니다. 컴파일된 결과는 바이크 코드 형태의 Contract입니다. 이 Contract를 geth에 배포하면 블록체인의 블록 형태로 저장되고 이후 EVM을 통해 실행이 됩니다. 이더리움이 P2P 이기 때문에 해당 Contract 는 다른 모든 이더리움 노드에도 복

세가 되어 실행이 됩니다. 이 과정을 잘 이해하려면 이더리움의 Account 개념을 잘 이해해야 합니다. 간략히 이해를 돕기 위해 설명하면 이더리움의 모든 기본 단위는 Account 입니다. 이더리움 지갑 등을 만들 때 실제 사람 사용자가 만드는 Account가 있고(EOAs , Externally Owned Accounts) , Contract Accont가 있습니다. 모든 Contract는 실제 Accont로 다뤄집니다. 이후에 좀 더 자세히 살펴보겠습니다.

이더리움 설치 및 프로그래밍 환경 꾸미기

1. 먼저 다음 3개의 오픈 소스 툴과 언어를 설치하여 사용 준비를 합니다.

-Brew : Mac용 패키지 관리자
-Go : Go-Ethereum 설치용 Go 컴파일로
-Geth : Version 1.6.7 (2017년 7월 30일 현재 최신 버전)

1) 먼저 Brew 를 설치합니다. 터미널에 다음의 명령어를 입력합니다.

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

2) 다음의 Go 공식 웹 사이트에 가서 Mac OSX 용 패키지를 다운로드 받아 설치합니다. 본 글에서는 1.8.3 을 사용합니다.

http://golang.org/dl

3) 다음으로 Geth을 설치합니다. 현재 Geth의 최신 버전은 V1.6.7 로 해당 버전을 사용합니다.

버전을 사용합니다..

다음에서 Geth의 소스코드를 다운로드 받는다.

- 다운로드 Geth 1.6.7.(ZIP)

또는 다음과 같이 git을 사용하는 방법도 있습니다.

git clone -b release/1.6.7 https://github.com/ethreum/gi-ethreum.git

다운로드 받은 해당 디렉토리로 이동 후 다음 명령을 실행하여 소스코드를 컴파일합니다.

$>> make geth

```
JAEHYUNui-MacBook-Air:go-ethereum-1.6.7  jaehyunpark-air$  make  geth
build/env.sh  go  run  build/ci.go  install  ./cmd/geth
>>>  /usr/local/go/bin/go  install  -ldflags  -s  -v  ./cmd/geth
```

4) 기타 환경 꾸미기

컴파일 후 "현재 설치한 폴더 밑에 /build/bin 폴더"에 geth 라는 실행 파일이 생성됩니다. geth 파일은 다른 라이브러리나 패키지에 dependency가 없기 때문에 원하는 폴더로 옮겨도 작동됩니다. 참고로 , 현재 Geth 소스코드는 Go언어로 개발되었기 때문에 컴파일시 Go 컴파일러가 필요하다.
Go1.8.3을 사용하여 컴파일 합니다.

* 참조 : https://github.com/ethereum/go-ethereum/wiki/Installing-Geth#build-it-from-source-code)

여러 버전의 Geth를 사용할 수 있기 때문에 편의상 alias를 만듭니다. 아래에

서는 /Users/jaehyunpark-air/go-ethereum/go-ethereum-1.6.7/build/bin/geth를 geth167로 alias합니다. 이후 geth167로 이더리움을 구동시킵니다.

```
JAEHYUNui-MacBook-Air:bin  jaehyunpark-air$  echo "alias geth167 ='/Users/jaehyunpark-air/go-ethereum/go-ethereum-1.6.7/build/bin/geth'  " >> ~/.bashrc
```

실제 쉘에 위의 내용을 반영을 해줍니다. 매번 로그인시 마다 적용을 위해서는 .profile 에 해당 내용을 반영해 둡니다. 편의를 위해 하는 작업입니다.

```
JAEHYUNui-MacBook-Air:bin  jaehyunpark-air$    source ~/.bashrc
```

이제 geth167 을 입력하면 바로 geth를 실행시킬 수 있다.

이더리움 엔진 갖고 놀아보기

이제 이더리움 엔진을 프라이빗 네트웍에서 구동시키고 다뤄보겠습니다. 가이드를 보면 프라이빗 블록체인을 구성할 때 4가지에 신경쓰라고 합니다.

- 커스텀 제네시스 파일 설정을 통해 최초의 이더리움 블록 생성
- 커스텀 데이타 디렉토리 설정을 통해 블록체인 스토리지 구성
- 커스템 네트웍 ID 설정을 통해 내가 구축한 프라이빗 블록체인 명명하기
- 프라이빗 네트웍에서 이용시 추천 사항으로 다른 노드와 연결하기 위해 자동으로 탐색하는 것을 방지하기

자 이제 슬슬 구동을 시켜 보겠습니다.

1)먼저 , 구동에 필요한 블록체인 데이타를 저장할 폴더로 privatechain을 생성한다. 원하는 이름으로 자유롭게 바꿔도 됩니다.

```
JAEHYUNui-MacBook-Air:bin  jaehyunpark-air$    mkdir privatechain
```

다음의 구동 명령을 통해 geth를 실행시킨다.

```
geth167 --identity "JayBlockChain" --rpc --rpcport "8080" --
rpccorsdomain "*" --datadir "/Users/jaehyunpark-air/go-
ethereum/go-ethereum-1.6.7/build/bin/privatechain" --port
"30303" --nodiscover --rpcapi "db,eth,net,web3" --networkid
1999 console
```

위의 구동 명령 중 --datadir 이 커스텀 디렉토리를 설정하는 옵션입니다. 그리고 커스텀 네트웍ID는 --networkid을 사용하여 1999로 설정하였고 --nodiscover 옵션을 지정하여 다른 노드에서 탐색하여 연결하는 것을 방지하였습니다. 이 설정을 하지 않으면 P2P 노드 연결을 위해 계속해서 ping 이 발생합니다.

```
--identity "JayBlockChain"    // 내 프라이빗 노드의 아이덴티티.
--rpc    // RPC 인터페이스 가능하게 함.
--rpcport "8080"   // RPC 포트 지정
--rpccorsdomain "*"   // 접속가능한 RPC 클라이언트 URL 지정 ,
                    // 가능한 *(전체 허용) 보다는 URL을 지정하는 게
                    보안상 좋음.
--datadir "/Users/jaehyunpark-air/go-ethereum/go-
ethereum-1.6.7/build/bin/privatechain" // 커스텀 디렉토리 지
정
--port "30303" // 네트웍 Listening Port 지정
--nodiscover    // 같은 제네시스 블록과 네트웍ID에 있는 블록에 연결
방지
--rpcapi "db,eth,net,web3"    // RPC에 의해서 접근을 허락할 API
--networkid 1999
  console   // 출력을 콘솔로 함.
```

참고로 마이닝이 가능하도록 구동시키려면 --mine 옵션을 설정해야 하는 데 이 설정이 작동되기 위해서는 미리 사용자 계정을 만들고 이 계정을 마이닝 작업 후 결과 Ether를 받을 Etherbase 설정한 후에 유효합니다.

`--mine  // 마이닝 모드로 구동 , Etherbase(coinbase) 설정 후 작동됨.`

다음은 geth의 Command line options에 대한 설명입니다.
* 참조 : https://github.com/ethereum/go-ethereum/wiki/Command-Line-Options

2) geth가 구동되었으면 이제부터 이더리움 콘솔상에서 자바 스크립트를 사용하여 작동시킬 수 있습니다. 이더리움 내부에 자바스크립트 런타임 환경을 구현했습니다. 먼저 Jay 라는 Account를 하나 생성합니다. 앞서 간략히 강조한 것처럼 이더리움에서 Account는 가장 중요합니다. 왜냐하면 모든 트렌젝션이 Account를 기준으로 작동되고 그 결과 Account의 상태를 바꾸는 방식으로 처리되기 때문입니다.

다음부터는 구동된 geth 의 Commnand line 상에서 하는 작업입니다.

// Jay Account 생성
```
> personal.newAccount("Jay")
> personal.newAccount("Jay")
INFO [08-01|08:46:29] New wallet
appeared                          url=keystore:///Users/jaehyunpark-a…
status=Locked
0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662
```

// 다음의 명령으로 계좌 목록 조회할 수 있습니다

> eth.accounts

```
> eth.accounts
["0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662"]
```

// 다음으로 마이닝 후 보상을 받을 이더베이스(etherbase)를 지정합니다. 여기서는 앞서 만든 jay 어카운트로 지정합니다.

> miner.setEtherbase(personal.listAccounts[0])

```
> miner.setEtherbase(personal.listAccounts[0])
true
```

personal.listAccounts[0] 는 0번째 Account를 말하는 것이고 , 다른 방법으로 Jay의 식별키를 지정해도 됩니다.

miner.setEtherbase("0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662")

//이제 계정을 만들고 이더베이스를 설정했다면 마이닝을 하겠습니다. 마이닝 후 리워드 보상은 앞서 지정한 Jay 어카운트로 보내집니다.

> miner.start()

```
> miner.start()
INFO [08-02|11:31:35] Updated mining
threads                    threads=0
Null
```

다음과 같이 마이닝 쓰레드를 2개로 줄수도 있습니다. - miner..start()

```
> miner.start(2)
INFO [08-02|11:31:49] Updated mining
threads                    threads=2
null
```

일단 실행을 시키면 백드라운드로 수행을 합니다. 이후 작업이 끝나면 결과를 콘솔을 통해 알려줍니다.

```
> INFO [08-02|11:37:14] Successfully sealed new
block                number=4 hash=4300e0…bd850f
INFO [08-02|11:37:14] 🔨 mined potential
block                    number=4 hash=4300e0…bd850f
INFO [08-02|11:37:14] Commit new mining
work                     number=5 txs=0 uncles=0 elapsed=683.934µs
```

마이닝이 성공적으로 수행되면 다음의 명령어로 실제 받은 보상 결과를 알 수 있습니다.

```
// 첫번째 계정의 잔액 조회
eth.getBalance(eth.accounts[0])
```

```
> eth.getBalance(eth.accounts[0])
300000000000000000000
```

위의 결과를 보면 Jay 계정에 300 Ether가 생성되어 있습니다. 위의 표시는 Wei 이기 때문에 $1/10^{18}$로 계산합니다.

```
// 생성된 블록 수도 조회해 볼 수 있습니다. 블록이 1개 생겨 있습니다.
eth.blockNumber
```

```
> eth.blockNumber
1
```

4) 프라이빗 네트웍에서 geth를 작동시키기 위해서는 먼저 커스텀 제네시스 파일을 생성한 후 이를 geth를 구동시 init 명령으로 함께 호출합니다. 제네시스 블록은 블록체인의 시작 블록이기 때문에 반드시 이를 만들어야 한다. 제네시스 블록을 만든 후 프라이빗 블록체인은 자유롭게 만들 수 있습니다.

제네시스 파일 생성은 아주 중요하기 때문에 아래 커스텀 제네시스 파일 생성시 문법과 체크가 엄격한 편이다. 이전 버전의 포맷과 문법이 작동안되는 경우가 많아 애먹을 수 있다. 문제가 생길 때는 기존 chain data를 삭제 후 다시 구동시켜 해결하면 된다.

```
//커스템 제네시스 파일 생성 , CustomGenesis.json
{
  "config": {
```

```
    "chainId": 15,
    "homesteadBlock": 0,
    "eip155Block": 0,
    "eip158Block": 0
  },
  "difficulty": "200000000",
  "gasLimit": "2100000",
  "alloc": {
    "0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662": {"balance":
"300000000000000000000"},
  }
}
```

위의 파일 설정 중 alloc은 마이닝 작업을 하지 않고서도 미리 해당 계정에 지정된 만큼 Ether를 할당하는 것 입니다. 여기서는 300Ether를 미리 할당 합니다.

다음과 같이 geth를 통해 CustomGenesis.json 파일을 구동시키고 제네시스 파일을 생성한다.

```
> geth167 --datadir /Users/jaehyunpark-air/go-
ethereum/go-ethereum-1.6.7/build/bin/privatechain init
CustomCenesis.json
```

```
JAEHYUNui-MacBook-Air:privatechain  jaehyunpark-air$    geth167 --
datadir  /Users/jaehyunpark-air/go-ethereum/go-
ethereum-1.6.7/build/bin/privatechain  init CustomGenesis.json
INFO [08-01|15:59:33] Allocated cache and file
handles          database=/Users/jaehyunpark-air/go-ethereum/go-
ethereum-1.6.7/build/bin/privatechain/geth/chaindata  cache=16
handles=16
INFO [08-01|15:59:33] Writing custom genesis block
INFO [08-01|15:59:33] Successfully wrote genesis
state          database=chaindata
                                                hash=d76c57…a31e33
INFO [08-01|15:59:33] Allocated cache and file
handles          database=/Users/jaehyunpark-air/go-ethereum/go-
ethereum-1.6.7/build/bin/privatechain/geth/lightchaindata  cache=16
handles=16
INFO [08-01|15:59:33] Writing custom genesis block
INFO [08-01|15:59:33] Successfully wrote genesis
```

```
state          database=lightchaindata
                                                    hash=d76c57…a31e33
```

이더리움 이더 갖고 놀아보기 : 송금.

이제 송금을 해 보겠습니다. 송금을 위해 Susie 라는 계정을 하나 더 만듭니다.

// Sueie 계좌 생성
```
> personal.newAccount("Susie")
```

```
> personal.newAccount("Susie")
INFO [08-01|16:15:36] New wallet
appeared                          url=keystore:///Users/jaehyunpark-a…
status=Locked
"0x87c4ef09c4e94249ed94b74d6d573c3dc902f15d"
```

어카운트를 조회해 보면 총 2개의 어카운트가 생성되어 있음을 확인할 수 있습니다.
```
> eth.accounts
["0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662",
"0x87c4ef09c4e94249ed94b74d6d573c3dc902f15d"]
```

// Jay -> Susie 로 1Ether를 송금을 해 봅니다.

```
> eth.sendTransaction({from :
'0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662', to :
'0x87c4ef09c4e94249ed94b74d6d573c3dc902f15d',
value:web3.toWei(1,"ether")})
```

위의 트렌젝션을 실행하면 Jay 계정에서 돈을 옮겨야 하는 데 LOCK되어 있으니 Unlock 시키라고 합니다.

```
>
personal.unlockAccount('0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662')
Unlock account 0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662
```

```
Unlock account 0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662
Passphrase:
 Error: could not decrypt key with given passphrase
```

다음과 같이 JAY 계정의 UNLOCK을 시킵니다.

```
>
personal.unlockAccount('0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662')
```

```
>
personal.unlockAccount('0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662')
Unlock account 0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662
Passphrase:
true
```

또는 다음과 같이 해도됩니다.

```
>
web3.personal.unlockAccount("0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662")
```

패스워드를 묻는 데 Jay 라고 계정 이름을 넣으면 됩니다. 대소문자 구별을 합니다. 다시 송금 트렌젝션을 수행하면 다음과 같이 잘 작동합니다.

```
> eth.sendTransaction({from : '0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662', to : '0x87c4ef09c4e94249ed94b74d6d573c3dc902f15d' , value:web3.toWei(1,"ether")})
INFO [08-01|17:03:49] Submitted transaction fullhash=0x265514bd911fbb8dab9cb54a7e5be57c35ce445cb07d1762cd0f0f61b78a5843 recipient=0x87c4ef09c4e94249ed94b74d6d573c3dc902f15d
"0x265514bd911fbb8dab9cb54a7e5be57c35ce445cb07d1762cd0f0f61b78a5843"
```

그런데 바로 트렌젝션이 수행되지 않습니다. Ether가 부족하거나 트렌젝션의 부정이 아니라는 것을 계산하는 등 정합성을 Mining을 통해 테스트를 해야 합니다.

다음으로 펜딩중인 트렌젝션을 확인할 수 있습니다.

>eth.pendingTransactions // 미확정 트렌젝션 확인

```
> eth.pendingTransactions
[{
    blockHash: null,
    blockNumber: null,
    from: "0xb2cf02bea7e2538a90b634fcbe2cbf1dd9ee6662",
    gas: 90000,
    gasPrice: 18000000000,
    hash:
"0x1bfdad2d242b36c5a4662c29b0edb919e4a539de3edeec876d8c8e9679c3b706",
    input: "0x",
    nonce: 2,
    r:
"0x4f9382cfb484c60565363d08a302d6e38e76cbd1ffb2877e840e2559348775e1",
    s:
"0x58f5e424ada1282d71cf9fead65777bbc86ff4884126a2442dd06c2c05c739ab",
    to: "0x87c4ef09c4e94249ed94b74d6d573c3dc902f15d",
    transactionIndex: 0,
    v: "0x41",
    value: 1000000000000000000
}]
```

다시 마이닝을 수행시켜 이더리움 블록내에 전달된 송금 트렌젝션의 정합성을 계산하고 문제가 없으면 체인내에 블록을 연결합니다. 이 연결이 실제 마이닝 작업 결과이자 송금이 완료되는 것 입니다.

>miner.start()

```
> miner.start()
INFO [08-02|12:09:22] Updated mining
threads                    threads=0
INFO [08-02|12:09:22] Transaction pool price threshold updated price=
18000000000
null
> INFO [08-02|12:09:22] Starting mining operation
INFO [08-02|12:09:22] Commit new mining
work                       number=5 txs=1 uncles=0 elapsed=31.313ms


> INFO [08-02|12:10:56] Successfully sealed new
block             number=5 hash=ae00c8…b8ca2b
INFO [08-02|12:10:56] 🔨 mined potential
block                      number=5 hash=ae00c8…b8ca2b
INFO [08-02|12:10:56] Commit new mining
work                       number=6 txs=0 uncles=0 elapsed=1.215ms
```

//다음으로 펜딩중인 트렌젝션을 확인할 수 있습니다.

>eth.pendingTransactions // 미확정 트렌젝션 확인

```
> eth.pendingTransactions
[]
>
```

//다음으로 Susie의 Account에 Ether가 송금됨을 확인할 수 있습니다.

`eth.getBalance(eth.accounts[1])`

```
> eth.getBalance(eth.accounts[1])
1000000000000000000
```

마치며

지금까지 Ether 콘솔상에서 personal , eth , web3 , miner 객체를 통해 이더리움 엔진을 다뤄봤습니다. 이를 이용하여 이더리움의 구동 환경과 실행 환경을 구축하며 , 이더리룸의 가상화폐인 Ether를 발행과 송금 등을 통해 이더리움 플랫폼에 대한 이해를 하였습니다. 다음에는 실제 `Contact` 프로그램과 간략히 작성하고 수행하면서 전체적인 이해를 완료하도록 하겠습니다.

오랜만에 다시 이것저것 엔진을 만지며 다시 여러가지 코드 조각을 만들어 보니 1998년 CORBA 엔진을 만들던 때가 떠오릅니다. 아마 당시에도 오픈소스 커뮤니티가 지금처럼 활성되었다면 더욱 크게 발전했을텐데 CORBA 자체의 개발 환경이 열악하다 보니 개발하는 데 많은 어려움이 많았습니다. 현재 이더리움도 초기 상태라 비슷한 상태로 보이는 데 조속히 멋진 개발 및 운영 도구들이 필요해 보입니다.