# Chapter 5. 블록체인의 핵심인 트랜잭션

블록체인에 기록이 남는 트랜잭션과 기록되지 않는 메시지를 이해한다. 함수의 호출을 표현하기 위해 기계 코드 수준의 ABI를 작성할 수 있다. 블록들은 서로 해시로 연결하고, 인증된다. 트랜잭션을 처리할 때 발생하는 수수료 gas가 필요하고, 명령의 종류와 데이터 량에 따라 계산할 수 있게 된다. 거래의 건수는 순서를 어긋나게 처리될 수 있기 때문에 추적하는 것이 필요하다.

# 1. 트랜잭션

블록체인에서의 거래(transaction)는 보통 금액이 이체되는 것을 의미하기 때문에, 입금계정과 출금계정이 필요하다.

온라인쇼핑과 블록체인의 거래를 서로 비교해 보자. 온라인 쇼핑에서는 로그인해서 상품 선택, 주문, 지급, 배송까지 하는 일련의 절차로 구성된다. 지급 버튼을 누르는 순간 거래가 발생한다고 하는데, 블록체인의 거래와 다를 바 없다.

다만, 지급에 필요한 금액의 형태와 송수신 내용이 블록체인의 것과 다를 수 밖에 없다.

온라인 쇼핑의 거래는 주문 내역이 텍스트 형태로 URL 인코딩 되고 (웹에서 인코딩되는 방식이다. 보통 인코딩과 달리 앞에 %가 붙기 때문에 퍼센트 인코딩이라고 부른다), 블록체인에서는 이진수로 인코딩된 바이트 코드로 전송된다. 인코딩하는 방식이 다를 뿐이다.

금액은 서로 실물 가치가 있는 금액이어야 한다. 온라인 쇼핑에서는 카드나 계좌 이체 등을 통해 금액이 지불된다. 블록체인에는 금액을 적는 'value' 필드가 있는데 거기에 적으면 된다. 또한 'gas'라는 비용이 발생한다. 카드 회사의 처리 수수료 또는 은행의 입출금 수수료에 해당된다.

거래 금액이 정해지면, 온라인 쇼핑이나 블록체인이나 송수신이 당연히 필요하다. 블록체인에서는 송수신 대상이 'address'라고 하는 암호화되어 해킹이 거의 불가능한 컴퓨터가 이해하는 16진수로 되어 있다는 점이 차이라고 볼 수 있다.

다음 표는

|  |  |  |  |
| --- | --- | --- | --- |
| 구성 | 온라인 쇼핑 | 이더리움 거래 | 비교 |
| 데이터 | 웹에서 작성된 주문 내역이 텍스트 형태로 구성됨 | **바이트 코드** | 블록체인은 EVM에서 처리할 수 있는 바이트 코드로 작성 |
| value | 은행, 카드, 페이와 연계된 금액 | Ether | 모두 실물 가치가 있는 금액, 단 블록체인에서는 'gas' 또는 'value' 필드에 적어서 전송 |
| 송수신 | 신용카드, 은행 계좌, 페이 | Address | 블록체인에서는 주소 'address'로 송금 (은행 없이) |

거래는 블록체인에 기록이 된다. 그렇기 때문에 블록체인을 안전한 매체라고 강조하고 있다고 설명한 바 있다.

기록이 되지 않는 경우 (외부에서 조회 전용 함수를 호출하거나 내부에서 스마트 계약간에 호출되는 경우) "**메시지 호출(Message call)**" 또는 **호출(call)**이라고 한다. 이 경우 **기록되는 거래(Transaction)와 구별**된다. 메시지 호출은 송신, 수신, 데이터, Value, gas 등의 항목을 가질 수 있다는 점에서 거래(트랜잭션)와 비슷하다. 거래는 **sendTransaction함수**를 사용하고, 블록체인에 기록된다.

다르게 설명하면, 거래는 계정의 서명이 필요한지에 따라 구분된다. **메시지 호출 call은 서명을 하지 않고**, **거래는 서명**을 해야 한다. 다시 말하면, 거래는 전송자에 의해 서명이 된 메시지를 말한다.

|  |  |  |
| --- | --- | --- |
| 함수 | 설명 | 블록체인에 기록 |
| call | 로컬에서 실행, 블록체인에 전송되지 않는다. 따라서 블록체인을 변경하지 않고, 읽기만 하는 경우, 비용이 발생하지 않음 | 기록되지 않는다. |
| sendTransaction | 외부계정에 의해 발생하고, 사인해서 블록체인에 전송된다. 블록체인을 변경할 경우 사용 | 기록된다. 반환값은 hash. |

## 1.1 call

call() 함수는 로컬 블록체인에서의 호출이라 블록체인에 전송되거나 기록되지 않는다. 예를 들어 거래의 내역을 조회한다고 하자. 이런 것은 call이고, 블록체인에 기록되지 않는다.

이 함수는 읽기 전용 호출이라서 블록체인의 상태에 변화를 줄 수 없다. 그래서 **사인이 필요없고**, gas가 발생하지 않는다.

블록체인에 기록되지 않기 때문에, 네트워크 참여자가 그 함수가 호출되었는지 알지 못한다.

call() 함수는 실행되고 나서 반환 값이 있을 수 있다. 프로그래밍에서 함수를 호출 한 후 값이 반환될 수 있는 것과 비슷하다.

web3의 API는 **web3.eth.call**이다. 차츰 배우게 된다.

## 1.2 sendTransaction

call() 함수와 대조적으로, sendTransaction() 함수는 블록체인에 전송되고 기록된다. **사인**해야 하고, gas가 필요하다.

트랜잭션은 **반환 값이 없다.** 정확하게 얘기하자면 반환값이 있기는 하지만, 그 거래의 해시 값이다. 합산을 한다고 가정하면, 합산 결과를 돌려받지 못하고, 대신 그 거래의 해시값이 반환된다.

프로그래밍을 할 때 반환 값은 함수가 올바르게 실행이 되었는지를 테스트할 때 유용하다. 블록체인에서 이러한 반환값이 없다는 것이 불편할 수 있다. 이럴 때 사용하는 방법은 로그(log)를 이용하는 것이다. 예를 들어 **반환 값을 알려면 event를 발생**시켜서 로그에 적어주고, 나중에 그 로그를 확인해야 한다.

해시 값이 주어지면, 트랙잭션은 마이닝이 되어야 한다. 거래가 인증이 되려면 마이닝이라는 과정이 반드시 필요하다.

web3에서 제공되는 API는 **web3.eth.sendTransaction**이다.

트랙잭션을 코드로 예를 들어보자. 지금은 코드만 확인하기로 하고, 나중에 실제 코딩하고 실행하게 된다.

줄1 coin.send(web3.eth.accounts[0],100,{from:web3.eth.accounts[0],gas:100000});  
줄2 coin.send.sendTransaction(web3.eth.accounts[0],100,{from:web3.eth.accounts[0],gas:100000});

인자를 입력할 경우, 줄1의 send() 함수에 적거나, 줄2의 sendTransaction() 함수에 적어도 된다. 적고 있는 항목은 다음과 같다 - (1) 수신: accounts[0], (2) 송신: accounts[0], (3) 송금액: 100 Wei, (4) gas 수수료: 100000 Wei.

더 알아보기: 로그

로그란 일반적으로 텍스트 형태로 파일에 저장되는 기록이다. 주로 서버 등에서 동작하는 프로그램들은 현재 어떤 일들이 발생하고 있는지 화면에 출력할 수 없는 경우가 많다. 그럴 때 지정된 로그 파일에 발생하거나 실행 과정 등을 기록하는 것을 로그라고 한다. 일반적으로는 파일에 저장되지만, 특별한 경우에는 지정된 화면이나 프로그램에 전달되어 출력된다.

# 2. ABI 명세

Application Binary Interface (ABI)는 컨트랙의 함수를 호출하기 위해 사용하는 방식이다.

개발자가 하는 것이 아니라, 프로그램 모듈 간의 인터페이스로 쓰이기 때문에 기계 코드 수준의 작동 방식이다. 프로그래밍에서 모듈의 인터페이스를 호출할 때 자주 사용하는 API, 단 **기계 코드 수준의 API**라고 이해하면 된다.

블록체인 외부에서나 또는 블록체인 내부에서 **컨트랙간에 서로 호출할 때, 함수 및 인자를 ABI 표준에 따라 부호화**해서 호출하게 된다.

이더리움에서 제공하는 예제를 통해 설명해보자 (https://solidity.readthedocs.io/en/develop/abi-spec.html). Foo의 baz()함수를 호출한다고 가정한다. 블록체인에서 함수와 인자를 각각 인코딩하고, 조합해서 16진수 데이터로 만들고, 호출하는 과정을 보면서 ABI를 이해해보자.

contract Foo {  
 function bar(fixed[2] xy) {}  
 function baz(uint32 x, bool y) returns (bool r) { r = x > 32 || y; }  
 function sam(bytes name, bool z, uint[] data) {}  
}

1. **함수명 부호화**

**함수 시그니처**, 즉 함수명과 인자를 적는다. 이때 인자는 **컴마로 분리하고 공백 없이** 적어 준다. 무심코 컴마 뒤에 공백을 넣지 않도록 한다.

함수 시그니처 baz(uint32,bool)를 **sha3 해시값**으로 만들고, **앞 4 바이트**를 선택한다. 이를 **function selector**라고 한다.

solidity 코드로 앞 4바이트를 구하면:

bytes4(bytes32(sha3("baz(uint32,bool)")))

이런 작업은 geth 콘솔 프롬프트 >에서 하면 바로 결과를 출력할 수 있다. 기억 나겠지만, \_gethNow를 작성하면서, 원격접속을 허용하는 API에 web3를 적어 넣은 적이 있다. 그 web3를 사용하게 되는 것이다.

geth 콘솔 프롬프트에서 해보자. 다음에 보인 것처럼 web3.sha3() 함수와 substring() 함수로 앞 4 바이트를 선택할 수 있다. 0x를 포함하므로 10개를 분리하면 된다.

geth> web3.sha3('baz(uint32,bool)').substring(0,10);  
"0xcdcd77c0"

1. **매개변수 값을 부호화**

매개변수가 69는 16진수로 바꾸면 "0x45", 1은 "0x1"이다.

geth> web3.toHex(69)  
"0x45"

**32 바이트**로 만들기 위해 **앞에 0으로 채운다**.

0x0000000000000000000000000000000000000000000000000000000000000045 - 인자 uint32 69 (32바이트)  
0x0000000000000000000000000000000000000000000000000000000000000001 - 인자 bool 1 (32바이트)

1. **데이터 조합**

위 1) function selector부호값과 2) 인자 부호값를 합친다. 두 번째 인자에서 16진수임을 나타내는 0x를 제거한다. 아래에서 볼 때 몇 줄로 나누어져 있지만 그것은 지면 문제이고, 한 줄로 쭉 연결된 문자열이다.

0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000000000000000000000000000000000001

1. **함수 호출**

함수를 호출할 때 data 항목에 위와 같이 생성된 ABI 부호를 적는다.

geth> web3.eth.sendTransaction({  
 from: eth.accounts[0],  
 to: "0x672807a8c0f72a52d759942e86cfe33264e73934",  
 data: "0xcdcd77c000000000000000000000000000000000000000000000000000000000000000450000000000000000000000000000000000000000000000000000000000000001",  
 gas: 400000}  
)

이 표준은 EVM에서 컨트랙의 함수를 호출할 때 사용하는 방식의 인코딩이기 때문에 복잡하다. 사람이 부호화되는 과정을 이해하고 코딩하기는 어렵다. 그러나 걱정하지 말자. 이더리움 내부에서 알아서 하므로 개발자가 직접 이 방식으로 코딩하지 않는다. 여기서는 내부적으로 이런 형태로 동작한다는 것으로 이해하자.

solidity의 **abi.encodeWithSignature()** 함수를 사용하면 ABI 부호화를 하지 않아도 된다.

<address>.call(abi.encodeWithSignature("baz(uint32,bool)",69,1))

또한 ABI를 사람이 직접 적어도 된다. 인자가 길어서 부분적으로 생략하고 보였다.

addr.call("0xcdcd77c0..생략..00001")

## 3. 트랜잭션 구성

## 3.1 비트코인의 트랜잭션

비트코인에서의 거래는 한 주소에서 다른 주소로 금액이 이체되는 것으로, 네트워크에 전파되고 모여서 블록으로 만들어지게 된다.

거래는 다음 표에서 보인 것처럼 버전, locktime 타임스탬프, 입금, 출금 관련 항목으로 구성한다.

|  |  |  |
| --- | --- | --- |
| 길이 | 항목 | 설명 |
| 4 bytes | Version | 버전정보 |
| 1-9 bytes | Input counter | in 개수 |
| variable | in | array로서 이전 거래의 out에서 가져와 본 거래의 out으로 지급된다.사용하게 되는 잔액을 가지고 있는 transactionHash, script를 가진다. |
| 1-9 bytes | output counter | out 개수 |
| variable | out | array로서 in보다 같거나, 적어야 한다 (적으면 처리비용으로). UTXO OpCodes: Scripts - 6...4사용하고 남는 잔액들 unspent transaction output UTXO. script, value를 가진다. |
| 4 bytes | locktime | unix timestamp 또는 블록번호 |

비트코인의 거래에서 중요한 항목은 (1) 입금 (in), (2) 수신 (out), (3) 금액이다. - in 항목에는 입금되는 주소와 잔액이 적힌다. 하나의 계정에서 모든 금액이 채워질 수도 있고 또는 여러 개가 필요할 수도 있다. - out 역시 하나 또는 여러 계정이 필요할 수 있다. 수신계정에 입금하는 거래가 이루어지고 나서, 잔액 역시 하나 이상의 주소로 환급되어야 할 수 있다.

비트코인에서는 송금하고 나서 잔액을 환급하는 특징을 가지고 있다. **하나 이상의 in에서 입금액을 채우고, 하나 이상의 out에 그 잔액을 환급**하게 된다. 따라서 **미사용잔액**이 적힌다.

아래 사례에서 보듯이, 입출금이 하나가 아니고 복수일 수 있다는 점에 주의한다.

거래에서 발생한 미사용 잔액이 수합되어야 하는 경우 input이 여러 개가 된다. ins는 이전 거래 outpoint 2건에서 미사용 잔액을 수합하여 output으로 지급한다.

출금 또한 수신자가 1명이라도, 그 잔액이 자신에게 돌려지므로 **최소 2개**가 된다. 하나는 수신처로 송금되는 금액, 다른 하나는 거스름돈에 해당하는 잔액이다.

Value는 Satoshi 금액이다 (1 BTC = 100,000,000 Satoshi).

>>> import bitcoin, pprint  
>>> tx = ''  
>>> tx\_structure = bitcoin.deserialize(tx)  
>>> pprint.pprint(tx\_structure)  
  
{'ins': [{'outpoint': {'hash': '5a4a44f45cb5653a770660833e747d0449dc91e5b02e815375702856bebe9c34',  
 'index': 3},  
 'script': '47304402203f02d209168bb9c4e8fab3b2f2aca19be1f9032eaa85be3a029641eecf69e60d022055d7e22914bb4741ee8c1ab73e23766de034a82dae22dad0c13684d6cb098b9e012102c3a7056ad278bf04f2e5546e29466d8175ec4a052ad8f9901b32d53e442545a8',  
 'sequence': 4294967295},  
 {'outpoint': {'hash': '386218d00f9617b55883499e96165c4e035cb7dcde4db6b7713995e18ae19008',  
 'index': 1},  
 'script': '4830450221009511497e136765037a75f75a126168502bfee73a7c286ea775968a7140368ef002200cfc776ade17b9444bb7ad13e71852974befd1d6fcfa196df677cd72a697c63b012102b8963213667c71471d62361fe27aad7ab2b8b102a6d93f7367634466e04b5422',  
 'sequence': 4294967295}],  
 'locktime': 0,  
 'outs': [{'script': 'a914ed91639c578b4c1f526525e6dcbc9fbc505fe76987',  
 'value': 47220000},  
 {'script': '76a9145f9a3e6e8029dd934a2a00a17aa8254c0975219988ac',  
 'value': 900433}],  
 'version': 1}

## 3.2 이더리움의 트랜잭션

이더리움에서는 미사용잔액을 수합해서 사용하는 방식이 아니라, 트랜잭션의 구성항목이 **입금주소 from**, **출금주소 to**와 **금액 value** 로 단순하다. 잔고를 확인하기 위한 비트코인에서의 미사용잔액이 없다.

거래를 수행하려면, 다음과 같은 필요한 속성이 제공되어야 한다. from에 20바이트 주소를 반드시 적어야 한다. from을 제외한 나머지 항목은 선택항목이라 적지 않아도 된다. 단 nonce는 동적으로 생성된다.

|  |  |  |
| --- | --- | --- |
| 속성 | 바이트 | 설명 |
| from | 20 바이트 | 전송 주소. 명시하지 않으면 web3.eth.defaultAccount |
| to | 20 바이트 | (선택) 수신 주소. 컨트랙 생성하는 경우는 당연히 명시하지 않아도 된다. |
| value | 32 바이트 | (선택) 전송 Wei 금액 |
| gas | 32 바이트 | (선택) 거래의 실행에 허용되는 최대 gas량 (미사용분은 반환) |
| gasPrice | 32 바이트 | (선택) 비용계산에 적용할 gas 가격 (Wei), web3.eth.gasPrice에 해당. |
| data | 제한 없슴 | (선택) 바이트 코드 |
| nonce | 32 바이트 | (선택) 계정에서 발생한 거래 순서 번호. 동일한 번호가 있으면 하나는 처리 거절, 또는 순서대로 처리 (즉 뒷 순서가 먼저 처리되고 앞 번호를 취소하는 이중 거래 편법을 막음) |
| v, r, s | 1, 32, 32 바이트 | v, r, s는 거래의 디지털사인(signature)에서 구하고, 공개키를 회복하기 위해 사용된다. v는 공개키를 회복하기 위한 1바이트 크기의 식별자이고, r은 개인키의 소유자에 의해 서명되었으며, s는 전송 중에 수정되었는지 검증하는 용도로 사용된다 (앞 4장에서 파이썬 eth\_keys 라이브러리를 활용하여 계산하고 있다) |

이더리움의 거래를 구성하려면 필드에 값을 넣어 다음과 같이 구성한다.

transactionObject = {  
 from: "0x...생략...A0A0A0A01",  
 to: "0x...생략...A0A0A0A02",  
 value: web3.eth.getBalance("0x...생략...A0A0A0A01"),  
 gas: "----",  
 gasPrice: "----",  
 data: "0x----",  
 nonce: 0,  
}

# 4. 트랜잭션의 처리 단계

거래가 발생하면, 블록체인에서 그대로 성립하는 것이 아니다. 그 거래는 마이너에 의해 인증되어야 한다.

갑이 을에게 송금 거래를 한다고 하자. 갑은 거래에 디지털 서명을 하고, 그 거래는 참여 노드들에게 전파된다. 노드들은 거래를 받으면:

* 거래의 **디지털 서명**을 인증하여 원본이 맞는지, 소유주의 서명이 맞는지,
* 전송자가 **gas를 지불할 잔고**를 가지고 있는지,
* gas가 **gas limit**을 초과하지 않는지 검증한다.

마이너 노드들은 이 거래들을 묶어 블록으로 만들어 가고, 블록이 채워지면 이른바 '합의'를 해야 한다. 합의된 블록은 체인에 연결이 되어 간다.

거래가 블록으로 만들어지는 과정을 단계별로 설명해 보면 다음과 같다.

|  |  |  |
| --- | --- | --- |
| 순서 | 단계 | 설명 |
| 1 | 거래 생성 | 거래에 **디지털 서명을**하고, 전송하기 위해 바이트 단위로 변환, **직렬화(serialize)**한다. |
| 2 | 거래 전송 | 거래가 p2p 네트워크로 전송되고, 피어 노드들에게 서로 전파된다 (broadcast). |
| 3 | 블럭 생성 | 일정 시간 동안 거래를 묶어서 블록이 될 때까지 **pendingTransaction**에 넣는다. 그러나 아직 블록체인에 연결되기 전이므로 후보 블록 **candidate block**이 된다. 마이너들이 이렇게 묶은 블록 후보는 네트워크로 전파된다. 마이너는 자신들이 거래를 수집하여 블록으로 만들기 때문에 서로 같지 않을 수 있다. 거래가 도착 순서대로 묶어지는 것이 아니라, **우선순위**가 높은 거래부터 시작해서 블록으로 만들어진다. 이더리움에서는 처리비가 높은 거래의 우선순위가 높다. |
| 4 | 블록 인증 | 블록을 인증하게 된다. 마이닝에 Proof of Work를 적용하면, **hash puzzle을 풀어 nonce를 정하는 작업**을 한다. 블록 해시를 가장 먼저 찾아낸 노드는 이를 네트워크에 알리고, 다른 노드들은 새로 생성된 블록의 유효성을 검사하고 인증한다. **가장 긴 길이를 가진 체인을 선택**하여, 블록을 앞 블록에 **전의 블록 hash를 찾아 체이닝**되어 블록에 추가된다. 채굴되었지만 체인에 연결이 거절된 블록은 그대로 유실되는 것은 아니고 나중에 블록에 추가된다. 참여한 노드들 중에서 가장 먼저 해시값을 찾아낸 노드가 보상을 받는다. 2016개마다 문제의 난이도가 조정된다. |
| 5 | 새로운 블록 전파 | 새로운 블럭이 공지되면 **다른 마이너들은 이 블록 해시값이 올바른지 검사하고 받아들이게 되면서 마이닝은 종료**하게 된다. 거래 기록은 중앙 서버가 가지고 있는 것이 아니라, 분산 네트워크에 실시간 공유된다. 공유되고 나면 수정할 수 없다. 로컬 블럭체인 동기화, 마이닝하지 않은 경우 동기화가 뒤떨어질 수 있다. |

# 5. 블록체인

## 5.1 블록헤더

### 블록헤더는 해시로 서로 연결된다.

블록은 헤더와 바디로 구분된다.

먼저 블록헤더를 보자. 가장 중요한 것은 현재 블록은 **전 블록의 해시**를 통해 연결되며, 이러한 연결이 블록체인을 구성하게 된다.

### 블록헤더에 포함되는 트리의 해시들

이더리움에는 4개의 트리가 있다. 여기서 트리는 trie라고 적히는데, tree와 같은 데이터 구조이다.

tree와 기능에 있어 다소 차이가 있어 발음을 트라이라고 해 구분하기도 한다.

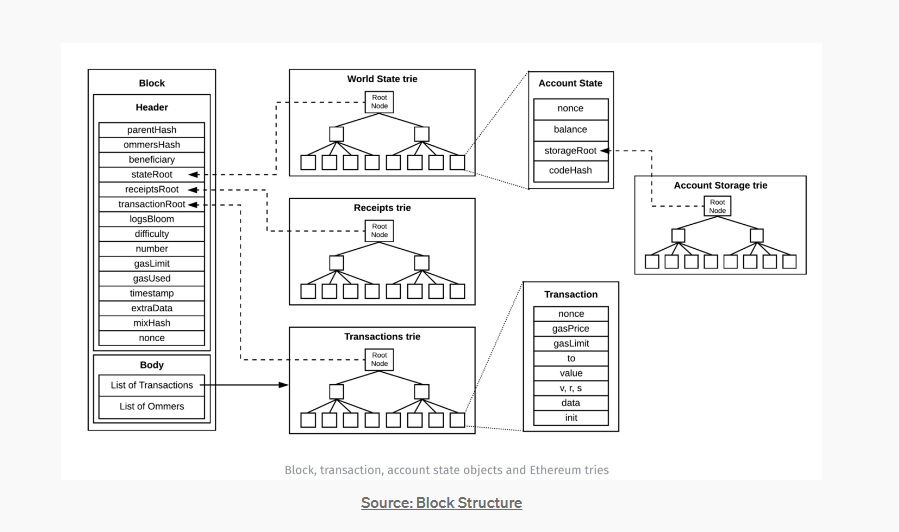
트리를 이해하면서 아래 그림과 같이 보도록 하자.

* 1. 거래 트리 **Transaction Tries**: 거래의 모음으로 트리구조로 만들어 지고, 블록 당 하나씩 존재한다. 블록에 있는 모든 거래 정보는 자료구조에서 학습했던 트리(tree) 구조로 구성된다. 블록체인에서 사용된 트리 구조는 랄프 머클(Ralph Merkle)이라는 사람이 개발했던 특별한 트리 구조로 주로 검증을 목적으로 사용된다. 각 거래정보들을 이용해서 해시값들이 계산되고, 다시 이들을 묶어놓은 최종 해시값이 만들어진다.
  2. 거래 수령 트리 **Transaction Receipt Tries**: 거래를 받았다는 증명을 모아서 트리구조로 만들어 지고, 각 블록 당 하나씩 존재한다.
  3. 저장 트리 **Storage Tries**: 모든 컨트랙의 계정이 하나씩 가지고 있다. 계정을 키로 사용하고 (그래서 키는 32바이트), 이러한 키별로 컨트랙의 상태 데이터를 저장하고 있다.
  4. 상태 트리 **State Trie**: 역시 계정 당 하나씩 존재한다. 앞서 설명한 바와 같이 계정에 대해 balance, nonce, storageRoot, codeHash를 가진다.

### 블록헤더는 다른 State, Transaction, TransactionReceipt Tries의 참조 값을 가진다.

블록은 다른 구조체에 대한 참조를 위해 다음과 같은 해시를 가지고 있다. 이것 역시 다음 그림을 보면서 이해하자.

* 앞서 설명한, 블록을 서로 연결하기 위해 전 블록의 해시(prevHash 또는 parentHash)
* 그리고 거래 트리의 해시 (블록은 **거래를 묶어 이진트리인 머클 트리(merkle tree)로 구성**해서 가진다)
* 거래 수령 트리(Transaction Receipt Trie)의 해시
* 상태 트리(State Trie)의 해시

 (그림에서 World State troie --> 상태 트리, Receipts trie -> 거래 수령 트리, Transactions trie --> 거래 트리, Block --> 블록, Header --> 헤더, Body --> 바디, List of Transactions --> 거래 목록, List of Ommers --> Ommers 목록으로 번역해주세요)

### 지난 블록의 해시를 가지고, 이를 통해 블록체인이 된다.

다른 트리와의 연결을 위해 가지고 있는 해시 외에도 블록체인이 되는 핵심인 이전에 생성된 블록해시 값이 다음 블록 헤더의 'parentHash' 항목에 저장된다. 각 블록에 트랜잭션이 지속적으로 쌓이면 새롭게 블록을 생성하여 **블록해시 값을 연결고리로 블록을 연결하여 블록체인이 생성**된다.

블록 뒤에 다른 블록이 추가될 때마다, 이를 **confirmation**이라고 하는데 confirmation이 늘어날 수록 수정이 힘들어 진다. **포크(fork)**는 **같은 블록 높이를 가지는 블록이 여러 개**, 부모가 여러 자식 블록을 가지는 경우 발생한다. 복수의 마이너가 거의 동시에 블록을 생산하는 경우이다.

매 **2016블록마다 난이도(difficulty)가 조정**된다.

더 알아보기: 블록 높이(block height)

블록체인에서 선행블록의 수를 말한다. 예를 들어 genesis block의 블록높이는 0이다.

### 그 외의 블록 헤더 항목들

그 외에도 채 굴속도를 조정하는 난이도(difficulty), 블록이 생성된 시간 (timestamp), 목표 값(Target Hash)를 맞추기 위해 사용하는 난스(nonce), 각 거래의 해시 값을 결합하여 계산하는 머클 루트의 해시값 (TransactionsRoot)으로 구성한다. 가벼운 클라이언트의 spv (simple payment verification)는 이러한 블록 헤더를 가지게 된다. 따라서 spv가 거래를 검증하기 위해서는 완전 동기화된 노드의 머클 트리를 조회해서 거래가 블록에 포함되었는지 확인하게 된다.

|  |  |
| --- | --- |
| 항목 | 설명 |
| parentHash | 이전 블록 해시 값으로 블록을 서로 체인으로 연결 (32바이트) |
| ommersHash | 블록내 ommers (또는 uncles) 의 해시 값. 동시에 블록이 2개 만들어질 수 있다. 이렇게 되면 어느 블록이 포함될 지 혼란스럽게 된다. 여기서 제외되는 블록이 ommer block(줄여서 ommer)이 된다. 부모의 형제를 의미해서 uncle이라고 불리웠지만, 중립적 ommer를 사용한다. |
| beneficiary | 현재 블록의 마이너 주소, 여기로 채굴 수수료가 전송된다. |
| stateRoot | 계정 상태를 가진 State trie의 루트 해시 |
| TransactionsRoot | 블록에 있는 모든 거래 hash의 최상위 루트 노드의 해시인 Merkle Root (32바이트). 따라서 어떤 블럭 해시 값이 변경되면 Merkle Root Hash값도 달라지게 되어 수정이 불가능함. |
| receiptsRoot | 거래영수증의 Root Hash |
| logsBloom | 로그에 포함된 색인정보 |
| difficulty | 4바이트 난이도를 조정하여 블록 생성 시간을 유지. |
| number | 블록의 순번. 참고로 genesis block은 0이고, 그 후 순서대로 번호가 매겨진다. |
| Gas Limit | 허용된 개스 한도. 이 한도에 따라 몇 건의 거래가 블록에 포함될지 결정된다.예를 들어, 5천의 gas가 필요한 거래A, 6천의 gas가 필요한 거래B, 3천의 gas가 필요한 거래C가 있다고 하자. gas limit이 11,000이라면 거래A와 거래B가 블록에 포함되고, 거래C는 포함될 수 없다. 거래C를 덧붙이려면 '한도초과' 오류가 발생하게 된다. |
| Gas Used | 블록에 포함된 모든 거래에 소비된 gas의 총량 |
| Timestamp | 마이닝이 끝나고 블록이 생성된 시간 (unix timestamp, 4바이트) |
| Extra Data | 별도로 저장하려는 데이터 |
| mixHash | 작업 증명에 사용되는 해시. 난스 값에서 생성. 목표 해시값(Target Hash)를 계산할 때, 헤더와 난스만 사용하는 것이 아니라 sha256(sha256(header + nonce + mixhash))에서와 같이 mixhash값이 포함된다. |
| nonce | 작업 증명 PoW에서 목표 해시값(Target Hash)를 맞추어 나갈 때 조정하는 값 (4바이트) |

geth 콘솔 프롬프트에서 특정 블록의 정보를 읽어보자. 위에서 설명한 항목들이 포함된 블록을 확인할 수 있다. getBlock() 함수에 현재 blockNumber 범위 내 임의의 번호를 적으면 된다.

geth> eth.getBlock(55169)  
{  
 difficulty: 6314813,  
 extraData: "0xd78  
 3010504846765746887676f312e372e33856c696e7578",  
 gasLimit: 4712388,  
 gasUsed: 21000,  
 hash: "0xd2c51...생략...94d26",  
 logsBloom: "0x00000...생략...00000", 전부 0으로 채워져 있다. 로그에 적힌 내용이 없다는 의미이다.  
 miner: "0x2e49e21e708b7d83746ec676a4afda47f1a0d693",  
 mixHash: "0x4da21d2c5696ee4d798f6445db0287a73fb2332f089ac43690d302827f52401a",  
 nonce: "0x3cd421957745a4b1",  
 number: 55169,  
 parentHash: "0x6533c71dd0f03108ea2c657896e73253f4f60ed3c4524a2bf0009cdec351d609",  
 receiptsRoot: "0x925b88d8f207cf48a9dc303902f5674d239072c1f03493febc311549d820b9ce",  
 sha3Uncles: "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",  
 size: 645,  
 stateRoot: "0xdbf466898fd964478d1f606e4a4dc1da9640af35f46d623d5ad4ed277526aaef",  
 timestamp: 1481249262,  
 totalDifficulty: 273764475970,  
 transactions: ["0xd87121b8b0f84f7fa038cd7c1928ca6a222d14228125c90edc2493fdef4fb90b"],  
 transactionsRoot: "0x5dbc80ced8364978e61d0c12aef1d7524092a4f7aa6ab30b4fccd6aa8b1282cd",  
 uncles: []  
}

## 5.2 블록바디

블록체인은 블록의 체인이고, 블록은 헤더와 바디로 구성된다고 설명하였다. 그렇다면, 블록 바디에는 무엇이 저장되어 있을까?

아주 간단하다. 블록바디는 발생한 트랙잭션을 저장한다. 그림에서 보듯이, 블록바디에는 **트랜잭션의 목록**, **ommer들의 목록**이 저장된다.

## 5.3 Merkle 증명

이더리움에서 블록은 거래 내용들을 해시화해서 트리 구조로 구성해서 가지고 있다. 머클 트리(Merkle tree)는 **hash로 이름이 붙은 가지 노드(lead node)로 구성된 트리**를 말한다.

머클 트리의 최정점에 해당되는 머클 루트는 가지들을 합쳐서 해시값을 구하고, 다른 노드에 대해서도 이 작업을 반복해서 결국 하나의 해시 값을 만드는 것이다.

예를 들어, 트랜잭션 T1, T2를 머클트리로 구성한다면 각 거래 데이터를 해싱(hashing)해서 나온 결과 값이 각 가지 노드에 저장된다. 부모 노드를 만들기 위해서 자식노드의 32바이트 크기의 트랜잭션 T1와 T2의 해시 값을 서로 연결해서 64바이트 문자열을 만들고 문자열은 **이중 해시** 처리되어 부모 노드의 해시를 생성한다.

최하층 노드에 대해 두 개씩 결합이 끝나면, 이런 식으로 그 다음 계층으로 올라가 반복해서 해시값을 구한다. 이러한 방식으로 상위에 노드가 하나 남을 때까지 계속 계산하고, 각각의 트랜잭션을 이진트리 형태로 만들 경우 가장 최종적으로 남는 해시 값이 머클트리 루트 해시 값이 된다.

블록 내에서 머클 해시 값을 통해서 **트랜잭션의 무결성**을 검증 할 수 있고, 블록 해시 값을 통해서 **해더 값에 대한 무결성**을 검증할 수 있다. 거래 내역의 위변조를 막기 위해서 해시로 만들고 이것을 트리 형태로 만든 것이다.

이렇게 해시로 연결해 놓으면 어떤 장점이 있을까?

특정 거래(transaction)가 수정되면, 그 해시가 변경되게 되고, **블록에 변조가 발생하면, 머클 트리의 루트까지 전달**된다. 즉 해시가 변경되는 것이 특정 거래가 있던 노드부터 시작해서 머클 트리의 루트 노드까지 발생한다. 이렇게 되면 머클 트리의 해시가 결국 다르게 되고, 블록 전체가 변조되기 때문에 위변조가 거의 불가능하게 된다.

### 머클 루트 구해보기

거래 txA, txB, txC, txD가 있다고 하자. 짝이 맞지 않으면, 홀수가 되는 거래는 자신을 복사해서 짝을 맞추어 연산한다. 머클루트(D)를 구하는 절차는 다음과 같다.

* (A),(C): 거래 데이터를 더블해싱, 바이트교환의 연산을 한다.
* (B): 위 (A),(C)에서 생된된 2개의 해시를 합쳐서 더블해싱을 한다.
* (D): 위 (B),(E)에서 생성된 2개의 해시를 합쳐서 더블해싱, 바이트교환의 연산을 한다. 이와 같이 2개씩 짝으로 연산을 거듭해, 최종 결과가 머클루트가 된다.

TxA--(해싱)-->hA--(바이트교환)-->hAswap (A)  
 |  
 (더하기)--(해싱)-->hAB (B)  
 | |  
TxB--(해싱)-->hB--(바이트교환)-->hBswap | (C)   
 |  
 (더하기)--(해싱)-->hABCD--(바이트교환)-->hABCDswap (D)  
 | 머클루트  
TxC--(해싱)-->hC--(바이트교환)-->hCswap |  
 | |  
 (더하기)--(해싱)-->hCD (E)  
 |  
TxD--(해싱)-->hD--(바이트교환)-->hDswap

물론 거래는 문자열로 구성되지 않지만, 다음과 같이 거래 데이터가 있다고 하자. 2개의 거래데이터 txA, txB를 묶어 머클루트를 구해보자. 거래데이터가 그 이상일 경우에도 반복적으로 연산을 하면 된다.

>>> import hashlib  
  
>>> txA = "Hello"  
>>> txB = 'How are you?'  
>>> txC = 'This is Thursday'  
>>> txD = 'Happy new Year'

### 해싱

먼저 거래 txA, txB를 해싱하자.

encode() 함수는 문자열을 바이트문자열로 변환한다. 아래에서 보듯이 앞에 byte string을 의미하는 b가 붙는다.

>>> "Hello".encode()  
b'Hello'

바이트 스트링에 대해 해싱을 하고, 또 더블 해싱(결과값을 다시 해싱하는 것)을 한다. 더블 해싱을 하게되면 해싱에 대해 공격 받지 않도록 좀 더 보안을 강화할 수 있다.

>>> \_hashA=hashlib.sha256(hashlib.sha256(txA.encode()).digest()).hexdigest()  
>>> \_hashB=hashlib.sha256(hashlib.sha256(txB.encode()).digest()).hexdigest()

digest()는 바이트 형식으로, 반면에 hexdigest()는 16진수로 출력한다.

>>> hashlib.sha256(txA.encode()).digest()  
b'\x18\_\x8d\xb3"q\xfe%\xf5a\xa6\xfc\x93\x8b.&C\x06\xec0N\xdaQ\x80\x07\xd1vH&8\x19i'

>>> hashlib.sha256(txA.encode()).hexdigest()  
'185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969'

### 바이트 교환

바이트 교환(Byte Swap)은 Endian을 변환하는 것을 의미한다. 빅엔디안(Big-Endian, BE)을 리틀 엔디안(Little-Endian, LE )으로 또는 반대로 변환한다.

문자열 HELLO의 예를 들어보자.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 구분 | H | E | L | L | O |
| 16진수 | 48 | 45 | 4C | 4C | 4F |
| BE 저장주소 (순서대로) | p | p+1 | p+2 | p+3 | p+4 |
| LE 저장주소 (역순으로) | p+4 | p+3 | p+2 | p+1 | p |

값이 뒤집히는 것이 아니라, 저장되는 주소만 변경되는 것으로 이해하면 된다. 즉 저장되는 메모리 순서가 BE와 LE 간에 차이가 있다.

* BE: 48-45-4C-4C-4F로 저장, 앞 글자가 먼저 빠른 주소에 저장, 즉 바이트 순서대로 저장
* LE: 4F-4C-4C-45-48로 저장, 뒷 글자가 먼저 빠른 주소에 저장, 즉 바이트 역순으로 저장

hello 문자열로 할 때는 뭐 그리 큰 차이가 없어 보인다. 그러나 00000000000000000000000000000000000000000000000000000000000000045와 같이 0이 많은 경우, LE에서는 45가 빠른 주소에 앞의 0은 다음에 저장된다.

이런 변환은 Endianess의 차이때문에 한다. 컴퓨터에서 동일한 CPU에서 작업이 일어날 때는 문제가 되지 않는다. 따라서 우리는 거의 이 차이를 인식하지 않고 데이터를 저장하고 불러오고 한다.

그러나 컴퓨터에서 다른 컴퓨터로 데이터를 반출할 경우 문제가 될 수 있다. **이더리움, 자바와 같은 가상머신에서는 BE**, **인텔 x86등 CPU 하드웨어에서는 LE**를 사용한다.

1 바이트, 즉 2 nibbles씩 출력해보자. 전체글자수 (즉 32바이트의 2배)에 대해 1 바이트인 2글자씩 반복을 하면서 출력한다.

>>> print(\_hashA)  
70bc18bef5ae66b72d1995f8db90a583a60d77b4066e4653f1cead613025861c

>>> for i in range(0, hashlib.sha256(txA.encode()).digest\_size\*2, 2):  
... print(\_hashA[i:i+2], end=" ")  
  
70 bc 18 be f5 ae 66 b7 2d 19 95 f8 db 90 a5 83 a6 0d 77 b4 06 6e 46 53 f1 ce ad 61 30 25 86 1c

반복문을 pythonic하게 줄여볼 수 있다. digest\_size는 바이트 크기이므로, 2를 곱해서 전체 크기를 구할 수 있다. 그리고 2바이트씩 잘라서 reversed() 함수로 뒤집는다.

>>> hashlib.sha256(txA.encode()).digest\_size  
32

>>> "".join(reversed([\_hashA[i:i+2] for i in range(0, hashlib.sha256(txA.encode()).digest\_size\*2, 2)]))  
'1c86253061adcef153466e06b4770da683a590dbf895192db766aef5be18bc70'

>>> hashAswap="".join(reversed([\_hashA[i:i+2] for i in range(0, hashlib.sha256(txA.encode()).digest\_size\*2, 2)]))  
>>> hashBswap="".join(reversed([\_hashB[i:i+2] for i in range(0, hashlib.sha256(txB.encode()).digest\_size\*2, 2)]))

>>> print("hashAswap: ", hashAswap, "\nhashBswap: ",hashBswap)  
hashAswap: 1c86253061adcef153466e06b4770da683a590dbf895192db766aef5be18bc70   
hashBswap: d0e7719d7633fb945d596090dfa31a95ae81b18d90352d63fc49af7f35ce2710

### 다음 단계의 해싱

#### 해시값을 결합한다.

앞서 구한 해시를 결합해서 다음 계층에서의 작업을 반복한다.

>>> hashAB=hashAswap+hashBswap

#### 또 해싱

더블해싱을 한다.

>>> hashlib.sha256(hashlib.sha256(hashAB.encode()).digest()).hexdigest()  
'e0c76d87a5a5c18ab29757603c5d1bda709306203b0a44c53fc6c90fba162903'

위 값을 변수에 저장한다.

>>> \_hashAB=hashlib.sha256(hashlib.sha256(hashAB.encode()).digest()).hexdigest()

### 또 바이트교환

바이트교환을 한다.

>>> "".join(reversed([\_hashAB[i:i+2] for i in range(0, 32\*2, 2)]))  
'032916ba0fc9c63fc5440a3b20069370da1b5d3c605797b28ac1a5a5876dc7e0'

위 값을 변수에 저장한다.

>>> hashABswap="".join(reversed([\_hashAB[i:i+2] for i in range(0, 32\*2, 2)]))

### 해싱하고 바이트 교환하는 함수 만들기

해싱하고 바이트 교환을 함수로 만들어 해보자.

def doubleHashByteSwap(raw):  
 import hashlib  
 size=hashlib.sha256(raw.encode()).digest\_size  
 \_hash=hashlib.sha256(hashlib.sha256(raw.encode()).digest()).hexdigest()  
 hashSwap="".join(reversed([\_hash[i:i+2] for i in range(0, size\*2, 2)]))  
 return hashSwap

>>> hA=doubleHashByteSwap(txA)  
>>> hB=doubleHashByteSwap(txB)  
>>> hAB=doubleHashByteSwap(hA+hB)

>>> print("hashA: {0}\nhashB: {1}\nhashAB: {2}".format(hA, hB, hAB))  
  
hashA: 1c86253061adcef153466e06b4770da683a590dbf895192db766aef5be18bc70  
hashB: d0e7719d7633fb945d596090dfa31a95ae81b18d90352d63fc49af7f35ce2710  
hashAB: 032916ba0fc9c63fc5440a3b20069370da1b5d3c605797b28ac1a5a5876dc7e0

# 6. 마이닝

## 6.1 인증

### 해시 맞추기

채굴이라고 불리기도 하는 마이닝은 거래를 인증하는 과정으로, 해시 퍼즐(hash puzzle)을 풀고 그 보상으로 새로운 코인을 생성하게 된다.

발생한 모든 거래가 추가되는 것은 아니다. **목표 값(target hash)을 찾아내는 계산**을 하고, 맞추는 경우에 인증되고 블록체인에 추가된다.

정답을 맞추는 작업을 하는 참여자들을 **채굴자를 의미하는 마이너**(miner)라고 하며, **보상은 정답을 맞춘 최초의 경우**에만 해당 마이너에게 주어진다.

이와 같이 일정량의 계산에 따라 거래가 인증되고, 블록체인에 추가될 수 있으므로 이러한 **계산 작업 증명을 Proof of Work** 이라고 한다.

비트코인의 경우 블록생성시간은 10분으로 설정되어 있으므로 하루 144 블록, 일주일에 1008블록이 생성된다. 이더리움은 문제의 난이도는 14초 내외에 1건씩 풀도록 정한다.

더 알아보기: 해시율(Hash Rate)

해시율(Hash Rate)는 1초에 해시(Hash)를 몇 회하는지 (Number of hashes per second, H/s) 마이닝하는 속도를 말한다. 성능이 좋은 컴퓨터는 당연히 Hash Rate이 높게 마련이다. H/s는 일반 척도와 같은 단위를 사용하여 나타낼 수 있다. 비트코인이 등장한 초기에는 보통 컴퓨터로 마이닝을 충분히 해낼 수 있었다. 채굴의 난이도가 높아지면서 특별히 마이닝을 하기 위한 **ASIC (Application Specific Integrated Circuit)** 칩을 사용한 특수 컴퓨터가 사용되고 있다.

|  |  |  |
| --- | --- | --- |
| 구분 | 단위 | 해시 회수 |
| Kilohashes | KH/s | 1000 |
| Megahashes | MH/s | 1,000,000 |
| Gigahashes | GH/s | 1,000,000,000 |
| Terahashes | TH/s | 1,000,000,000,000 |
| Petahashes | PH/s | 1,000,000,000,000,000 |

더 알아보기: **Ethash**

이더리움이 구현한 PoW 알고리즘이다. 이전에 사용하였던 Dagger-Hashimoto 알고리즘을 개선한 버전이다. Ethash에서는 "sha3\_256", "sha3\_512" 해시함수가 사용된다.

Ethash는 Ethash DAG라고 하는 1 GB 데이터 분량을 사용한다. 전체를 가지고 있을 필요가 없는 light client는 16 MB 캐시를 사용한다. 이 데이터는 매 **30,000 블록마다 재생성**되는데, 이 시간을 **epoch**이라고 한다. 마이닝할 때 DAG의 일부를 가져다가 블록해시와 nonce를 사용해 target hash를 맞춰 나갈 때 사용한다.

마이닝을 하기 전에 보상이 지급되는 계정 coinbase를 정해야 한다. coinbase를 충전할 계정으로 변경하고, reward가 주어질 계정으로 마이닝을 시작한다. **처음 채굴을 시작하면, 기가규모의 블럭체인을 내려받아야 한다. 약 10분 정도 소요되며, DAG가 구성**된다.

> miner.setEtherbase(eth.accounts[2])  
true  
> eth.coinbase  
"0x53cbba17cf9bd0735855809bdcb88e232de96f32"  
> miner.start();

### 체인에 연결

2개의 블록이 동시에 채굴되는 경우, 하나만 체인에 합쳐지고 나머지는 거절된다. 이때 거절되는 블록을 **고아(Orphan) 블록**, 즉 **부모가 없는 블록**을 말한다. 부모로 연결하려 했던 블록에 연결이 실패하면서 부모가 없어지는 경우에 발생한다.

비트코인에서 고아라고 불리우는 이들을 이더리움에서 **Ommer**로서 블록에 포함하고 있다. 고아블록과 Ommer는 거절된 블록이라는 점에서 비슷하지만, 이더리움에서는 Ommer 블록을 찾으면 보상을 하고 있다는 점에서 차이가 있다.

**스테일(Stale) 블록** 블록은 **자식이 없는 블록**을 말한다. 마이닝은 되었지만, 블록높이가 동일한 다른 블록이 먼저 자식이 생기면서 정상적인 블록체인이 되는 경우 발생한다. 고아블록에 자식으로 붙게 되면 Stale 블록이 된다.

이더리움에서의 합의 알고리즘은 GHOST (Greedy Heaviest Observed Subtree, Zohar and Sompolinsky in December 2013)이다. 인증이 되면서 블록에 포함되지 못하는 분기가 발생할 수 있다. 이 경우 GHOST의 Heaviest가 의미하듯이 가장 무거운 트리, 가장 많은 계산 노력이 투입된 것이 체인에 연결되는 방식이다. 비트코인은 가장 긴 블록을, 반면에 이더리움은 가장 무거운 블록을 연결한다.

## 6.2 난이도

**해시 퍼즐(hash puzzle)**은 **블록 해시(block hash)**값을 결정하기 위해 풀어야 하고, 난이도에 따라 얼마나 어려운지 설정된다.

난이도는 식으로 표현하면, **sha256(sha256(data+nonce)) < hash\_difficulty**이다. \* 식의 좌측은 데이터와 nonce의 더블해시 값을, \* 식의 우측 hash\_difficulty는 현재의 난이도가 반영된 목표 해시이다.

이 식이 충족되도록 nonce를 찾아야 한다. 그 알고리즘을 적으면 다음과 같다. 즉, 블록헤더의 Hash값이 난이도 목표에 제시된 값 hash\_difficulty보다 작은값이 나오게 하는 Nonce값을 찾는 것이다.

loop  
 if sha256(sha256(data+nonce)) < hash\_difficulty  
 stop  
 else change the nonce

예를 들어, **10분에 문제를 풀기를 기대했는데 5분에 풀었다**. 그러면 **난이도는 , 2**가 된다. 그러면 **새로운 난이도 = 현재 난이도 x 2**이 된다. 즉 난이도가 2배로 증가하게 된다. 난이도가 1보다 크게 되면 새로운 난이도는 증가하고, 반대는 감소하게 되는 방식이다.

**새로운 목표 해시값 = 이전 목표 해시값 / 난이도** 즉, 난이도가 높아지면 새로운 목표 해시 값이 낮아지고 맞추기가 더 어려워지게 된다.

예를 들어:

* 1부터 10 범위에 들어가는 수를 생성하는데 1분이 걸린다고 하자.
* 목표 값을 5로 정하고, 그 이내의 값이 나오려면 즉 2분이 걸린다 (1~5의 사이이므로 2배의 시간)
* 목표 값을 3으로 정하면, 즉 3분 20초가 걸린다.

즉:

* 목표 값이 최대값에 가까우면 쉬워진다. 그 보다 작은 수를 찾는 것은 당연히 쉽게 된다. 예를 들어 (**최대 32바이트 해시값은** , 이 보다 하나 적은 블록 hash값을 찾는 것은 쉽다.
* 최대 값과 목표 값의 간극이 벌어질수록 어려워진다.
* 비트코인에서 difficulty는 2016개의 평균시간이 1,209,600초 (2주)되도록 정한다 (10분에 1개씩, 1주일에 7일 x 24시간 x 6개 = 1008개).

그 절차를 좀 더 자세하게 설명하면:

* 1. 거래를 수집하여 **블록을 만든다** (비트코인은 1MB 블록). 블록에 있는 모든 거래의 hash 합으로 Merkle Root를 계산한다.
  2. 블록헤더 (Version + Previous Block Hash + Merkle Root + Timestamp + Difficulty Bits + Nonce) 값을 SHA-256 해시하고 또 재해싱를 하여 **해시를 계산**한다.
  3. 2에서 계산된 해시값을 **목표 해시값(Target hash)와 비교**한다. target hash보다 적으면 정답, 아니면 2번으로 돌아가 nonce값을 증가함.
  4. 문제를 푼 경우, 블럭체인에 승인된 블럭을 맨 뒤에 첨부, 모든 참여자들에게 공지하고 참여자들로 하여금 계산을 하게 하여 검증되면 **합의**한다.
  5. 참여자가 많아져 계산속도가 빨라질 수 있지만 설정된 블럭생성시간에 맞추어 **난이도가 조정**된다. 파이썬에서 //는 정수 나누기 연산이다.

block\_diff = parent\_diff + parent\_diff // 2048 \*   
max(1 - (block\_timestamp - parent\_timestamp) // 10, -99) + int(2\*\*((block.number // 100000) - 2))

즉, 부모블록과 현재블록의 생성시간의 차이에 따라 난이도 변경이 이루어진다. parent\_diff // 2048는 난이도 변경범위 (bound divisor of the difficulty)이다.

\* \*\*10초 이하\*\*: 난이도 \*\*상향\*\*, ```parent\_diff // 2048 \* 1``` 만큼 상향  
\* \*\*10~19초\*\*: 난이도 변동 없슴.  
\* \*\*20초 이상\*\*: 난이도 \*\*하향\*\* (timestamp 차이에 따라)최소 ```parent\_diff // 2048 \* -1```에서 최대 ```parent\_diff // 2048 \* -99```

* 1. 블록이 한꺼번에 2개 마이닝 되는 경우, 두 마이너가 동시에 풀면 fork가 발생했다고 하고, 난이도가 높은 블록이 인증되고 체인에 연결된다. 난이도가 낮은 블록은 엉클 블록(고아 블록)이 된다.

### 마이닝하면서 nonce를 정하는 연습

블록헤더 데이터의 해시 값에 NONCE를 증가시키면서 앞 자리의 0의 개수를 맞출 때까지 반복한다.

찾고자 하는 해시가 0000로 시작한다고 하자. 그러면 최대값은 0000FFFF...가 되겠다 (공간제약으로F를 모두 표현하지 않았다). 즉 그 해시의 최대 값보다 작은 범위 내에 들게 된다. 그러면 멈추고, 그 값을 해시로 정하게 된다.

비교할 목표 해시값의 앞 자리 숫자를 정하기 위해서, 숫자에서 1을 제외하고 0만 세어보자. 예를 들어 10000에서 1을 제외하면 0000이 비교할 앞자리 숫자가 된다.

>>> nzeros=4  
>>> str(pow(10,nzeros))[1:nzeros+1]  
'0000'

자릿수에 따라 난이도가 얼마나 영향을 받는지, 함수 mining()을 작성해서 실행해보자.

def mining(nzeros):  
 import hashlib  
 found=False  
 blockNumber=54 # hex  
 NONCE=0  
 data='Hello'  
 previousHash='5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e'  
 while found==False:  
 z=str(blockNumber)+str(NONCE)+data+previousHash  
 guessHash=hashlib.sha256(z.encode('utf-8')).hexdigest()  
 # e.g. nzeros=4 -> str(pow(10,nzeros))[1:nzeros+1] -> "0000"  
 if guessHash[:nzeros]==str(pow(10,nzeros))[1:nzeros+1]:  
 found=True  
 NONCE+=1  
 if(NONCE%pow(10,nzeros)==0): #print guessHash every 10000000  
 print("NONCE: ",NONCE, guessHash)  
 print("Solved ", "NONCE: ", NONCE, "guessHash: ", guessHash)  
 return guessHash

인자를 4, 5, 6으로 증가하면서 함수를 실행하면:

* 앞자리 0이 4개일 경우, 94280회 반복 Solved NONCE: 94280 guessHash: 000043ce4a61d02...7a13f
* 앞자리 0이 5개일 경우, 315753회 반복 Solved NONCE: 315753 guessHash: 000007f9f69a43f...df4d1
* 앞자리 0이 6개일 경우, 45576417회 반복 Solved NONCE: 45576417 guessHash: 0000003d02b9560...9994e

즉 Target Hash가 적어질수록 더 어려워지는 것을 알 수가 있다.

>>> mining(4)  
  
NONCE: 10000 e0458f4777612b5565be6df81b53f76943dac3f77939e886e0dc5a2fb5ad1855  
...생략... 94280회가 실행되고 나서 hash를 맞추고 있다.  
Solved NONCE: 94280 guessHash: 000043ce4a61d02bff0e68ba18a7daf448cb3b93691fdd4850f6cd3f85b7a13f

>>> mining(5)  
NONCE: 100000 218b6da50df07b86cb603e3ca2e1928381839996c426298dd89b2c98828eda19  
...생략... 315723회가 실행되고 나서 hash를 맞추고 있다.  
Solved NONCE: 315753 guessHash: 000007f9f69a43f1bb6ab92672d873b93d6bafaa2007e44b6151bd19efadf4d1

>>> mining(6)  
NONCE: 1000000 83c442aba508ff40f3abcb5395954b2507a514840a07ca6069b126de59aa057c  
...생략... 45576417회가 실행되고 나서 hash를 맞추고 있다.  
Solved NONCE: 45576417 guessHash: 0000003d02b95604bb1ec436ff20e08168dd339f2ec0f9941bfc58bad039994e

마이닝이 안되는 경우가 있을 수 있다.

* 발생한 트랜잭션이 처리되지 않고 여러 건이 적체되어 있는 경우,
* 처리비용이 낮은 경우,
* 또는 트랜잭션 nonce 앞 번호가 처리 되지 않았는데 뒤 번호가 대기하고 있는 경우, 뒤 트랜잭션은 대기하게 된다.

마이닝이 적체되어 안 지워지는 경우에는 바람직하지 않지만, 강제로 정리해야만 하는 순간이 있다. 이럴 경우에는:

* geth를 종료하고,
* datadir 아래 geth디렉토리 밑의 transactions.rlp를 삭제하고,
* geth를 다시 시작한다.

따라서 마이닝을 할 경우, txpool.inspect 명령어를 실행하여 적체되어 있는 거래가 있는지 확인하는 것이 바람직하다.

geth> txpool.inspect  
{  
 pending: {},처리할 수 있는 대기 거래 건수   
 queued: {} nonce가 순서에 맞지않아 처리할 수 없는 거래 건수  
}

# 7. gas

분산 시스템에서는 참여 노드의 CPU, 메모리 등의 컴퓨팅 자원을 필요하게 된다. 거래를 처리하는데 필요한 컴퓨팅 자원을 **gas**라는 단위로 표현하고, 여기에 **gasPrice** 단가를 곱해서 비용을 계산한다.

비용 = gasPrice \* gas

주유 요금에 비유하면 이해가 쉽다. 서울-부산을 주행한다고 하자. 주행거리에 따라 필요한 연료량이 gas, 단가는 gasPrice에 해당하고 이를 곱하면 주유 요금이 계산될 수 있다.

gas는 데이터 크기와 소요되는 컴퓨팅 자원량에 따라 산정이 된다. 예를 들어 Kecchak hashing을 하는데 30 단위의 gas가 필요하다. 위 산식의 gasPrice는 일종의 단가로서 gas에 곱해서 비용이 결정된다.

거래를 전송하는 코드에 필드로서 gas, gasPrice를 선택적으로 아래와 같이 적어주게 된다.

web3.eth.sendTransaction({from: , to: , gas: , gasPrice: })

## 7.1 gas 가격

gasPrice는 단가로서 사용자가 정할 수 있다. 주유와 비유하면 gasPrice는 시가, 즉 몇 ether가 필요한지에 해당한다.

보통 명시하지 않으면 최선의 가격으로 정해진다. 따라서 gasPrice를 정하지 않고, 거래를 전송하기도 한다.

그러나 gasPrice에 따라 처리속도가 결정된다는 점을 주의한다. - gasPrice가 너무 적으면 마이닝되지 않을 수 있으며, - 반대로 많으면 빠르게 마이닝 될 수 있다. gasPrice가 높을수록 더 많은 마이너가 처리하려고 들 것이고 거래가 마이닝되는 시간이 짧아지게 된다.

평균적으로 지불해야 하는 가격을 알려주는 사이트가 있다. 메인네트워크에서 gasPrice 단가는 시간에 따라 정해져 있지 않고 변동된다 etherscan.io에서는 지난 수 년간의 gasPrice를 그래프로 제공하고 있는데 (https://etherscan.io/chart/gasprice), 그 가격이 약간씩 감소하고 있지만 이는 이더리움의 실물가치가 상대적으로 증가하면서 생기는 현상일 수 있다.

또한 가격에 따라 채굴속도에 영향을 미치게 되는데, 예를 들어, 2분 이내로 빠르게 처리되려면 ('Fast'), 더 많은 가격을 지불해야 하겠다 (https://ethgasstation.info/)

gas price는 web3.eth.getGasPrice() 함수를 사용하여 구할 수 있고, 개인망에서 그 값은 1 gwei를 출력하고 있다. 보통 테스트네트워크는 마이닝의 우선순위에 영향을 미치지 않기 때문에, 고정 gas price를 가지고 있다.

gasPrice()를 계산해보자.

geth> eth.gasPrice  
1000000000

## 7.2 gas 한도

### 거래의 gas 한도

거래를 처리하기 위해서는 gas가 필요하며, 그 한도가 있다. 즉 최대 소모될 수 있는 gas 량이 한도가 된다.

그렇다고 사용자가 무턱대고 지불하지 않고, gas 한도는 사용자가 최대로 지불할 용의가 있는 한도를 말한다.

한도의 범위에서 gas가 소진될 때까지 실행하고, 남으면 반환된다. 이 한도를 낮게 설정해서 모자라면 **Out of Gas Exception**이 발생하고, 실행이 보장되지 않을 수 있다.

블록체인에 부담이 가는 실행 (예: 무한반복) 역시 gas 한도를 넘어가게 되면 가능하지 못하게 된다. 또한 과도한 gas를 사용하는 악의적인 시도 역시 예방할 수 있다. 이와 같이 거래에 비용이 발생한다고 하자. 대규모 거래가 수반되는 DDos 공격은 상당한 비용이 발생할 수 밖에 없어 쉽게 시도하지 못하게 되는 효과가 있다.

일정 gas가 사용되면 gasLimit을 초과하게 되므로 정지하게 된다.

### 블록의 gas 한도

또한 gas 한도는 블록에 대한 한도를 의미하며, 이는 블록에 포함될 거래의 갯수를 결정하기 위해 사용된다. 예를 들어, 거래가 6개 있고 gas비가 각 25, 30, 35, 40, 45, 50이라고 하자. 블록의 거래 한도는 100이라고 하자. 그러면 거래의 gas가 25 + 30 + 35를 포함할 수도, 40+45가 포함될 수도 있다.

지금 현재 9,994,671 gas이고 (https://ethstats.net/) 이를 21000(거래 gas limit)으로 나누면 475개의 거래가 한 블록에 포함되는 수준이다.

이 속도로 평균 14초 내외로 1블록이 생성되고 있다 (https://ethstats.net/). 블록에 대한 gas limit은 처음에 genesis.json에 설정되어 있고, 블록마다 이 값은 마이너들에 의해 1/1024 만큼씩 조정될 수 있다. 블록을 생성하면서 그만큼의 한도로 조정은 할 수 있으나, 큰 차이로 변경하는 것은 허용되지 않는다. eth.getBlock('latest').gasLimit은 현재의 gasLimit을 알 수 있다.

geth> eth.getBlock('latest').gasLimit  
8000000

## 7.3 gas 계산

스마트 컨트랙을 블록체인에 배포하거나, 뭔가를 저장하거나, 함수를 호출하는 경우 gas 비용이 발생한다. 실행비용과 거래비용으로 구분하여 산정하는 기준을 알아보자.

### 실행비용

컨트랙을 실행하기 위해서 어떤 동작이나 연산이 당연히 필요하고, 실행비용이란 이 때 지불해야 할 gas 비용을 말한다.

**실행 비용(execution costs)**은 사용되는 명령어 opcode를 기준으로 산정이 된다. 몇 가지 비용을 알아보면 다음과 같다.

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | 명령어 | Gas | 설명 |
| 00 | stop | 0 | 실행중지 |
| 01 | ADD | 3 | 더하기 |
| 10 | LT | 3 | less than 비교 |
| 20 | SHA3 | 30 | Keccak-256 hash 계산 |
| 30 | ADDRESS | 2 | 현재 실행 계정 주소 읽기 |
| 31 | BALANCE | 400 | 잔고 읽기 |
| 40 | BLOCKHASH | 20 | 블록 해시 |
| F0 | CREATE | 32000 | 컨트랙 생성 |
| FF | SELFDESTRUCT | 5000 | SUICIDE 실행 |

더 알아보기: Opcodes

Solidity는 사람이 읽을 수 있는 수준의 코드이다. 반면 기계어는 자신이 사용하는 명령코드로 실행되는데, 이를 **Op**eration **Code**를 말한다 (참조 https://ethervm.io/). 예를 들어, 스택에 2 3 ADD라고 저장이 되어 있고 (push), 하나씩 꺼내 (pull) 2 + 3 연산을 한다.

* 2 3 OP\_ADD 5 OP\_EQUAL
* 2 3 OP\_ADD 2 OP\_MUL 1 OP\_ADD 11 OP\_EQUAL

비트코인도 이러한 스크립트 언어를 사용하는데, 서명과 관련된 다음 스크립트도 Opcode로 실행된다. \* scriptPubKey는 공개키로 잠그는 opcode 명령문은 DUP HASH160 <PubHash> EQAULVERIFY CHECKSIG \* scriptSig: 는 반대로 공개키로 푸는 명령문은 <sig> <PubK> 이들 코드 scriptSig, scriptPubKey를 합쳐서 실행하면 P2PKH Pay-to-Public-Key-> Hash: \* OP\_DUP OP\_HASH160 OP\_EQUALVERIFY OP\_CHECKSIG

### 거래비용

거래 비용(transaction costs)은 블록체인에 컨트랙의 바이트 코드를 보내면서 발생한다. 이더리움 황서(Yellow Paper)의 부록 G에 산정되어 있다.

* 모든 거래의 기본 비용 ()는 21,000,
* 컨트랙을 배포하면서 생성하는 () 비용은 32,000,
* 데이터 또는 코드의 바이트 수에 따라 산정되는 비용은:
  + 1. 데이터 값이 없는 는 4,
    2. 데이터 값이 있는 는 68로 계산된다.

더 알아보기: 이더리움 백서(White Paper)와 황서(Yellow Paper) 이더리움 백서는 이더리움을 만들었던 비탈릭 뷰테린이 작성한 개론서라고 할 수 있다. 이더리움에 대한 비전부터 프로젝트에서 계획하고 있는 것들에 대해서 설명한다. 이더리움 황서는 가빈 우드가 작성한 이더리움의 설명서이다. 백서에 비해 좀 더 기술적이고 과학적인 내용들로 채워져 있고, EVM에 대한 사양 등이 명시되어 있다.

아래와 같은 트랜잭션 데이터가 있다고 하자. 길어서 중간 부분을 생략하고 있다.

txdata = "606060405260405160808061067283398101604090815281516020830151918301516060909301519092905b42811161003457fe5b60008054600181016100468382610100565b916000526020600020900160005b8154600160a06...0000000000007c6b393dbb1152691b02a61378bc731662cdd9f1000000000000000000000000748ed23daa18226d872b5bd4d48ff2594fd6901c00000000000000000000000000000000000000000000000000000000597fc480"

2자리씩, 즉 1바이트씩 읽어서 0인지 아닌지 그 갯수를 산정한다. count\_zero\_bytes() 는 데이터가 없는, 즉 0인 바이트의 수를 세는 함수이다.

def count\_zero\_bytes(data):  
 count = 0  
 for i in range(0, len(data), 2):  
 byte = data[i:i+2]  
 if byte == "00":  
 count += 1  
 return count

다음은 0이 아닌 데이터의 바이트 개수를 세는 함수이다.

def count\_non\_zero\_bytes(data):  
 return (len(data) // 2) - count\_zero\_bytes(data)

그 결과를 출력하면 0은 184바이트, 아닌 경우는 1594 바이트가 된다. 따라서 거래비용은 로 계산한다.

>>> print("zero-bytes: {0}".format(count\_zero\_bytes(txdata)))  
>>> print("non-zero-bytes: {0}".format(count\_non\_zero\_bytes(txdata)))  
zero-bytes: 184  
non-zero-bytes: 1594.0

### gas 산정 함수

거래를 실행하기 위해, 꼭 gas를 산정해야 하는 것은 아니다. 개스비용을 적지 않으면, web3.eth.estimateGas() 함수로 산정한 값으로 적용하게 된다. 이 함수는 나중에 사용해보자.

### 사용된 gas 알아보기

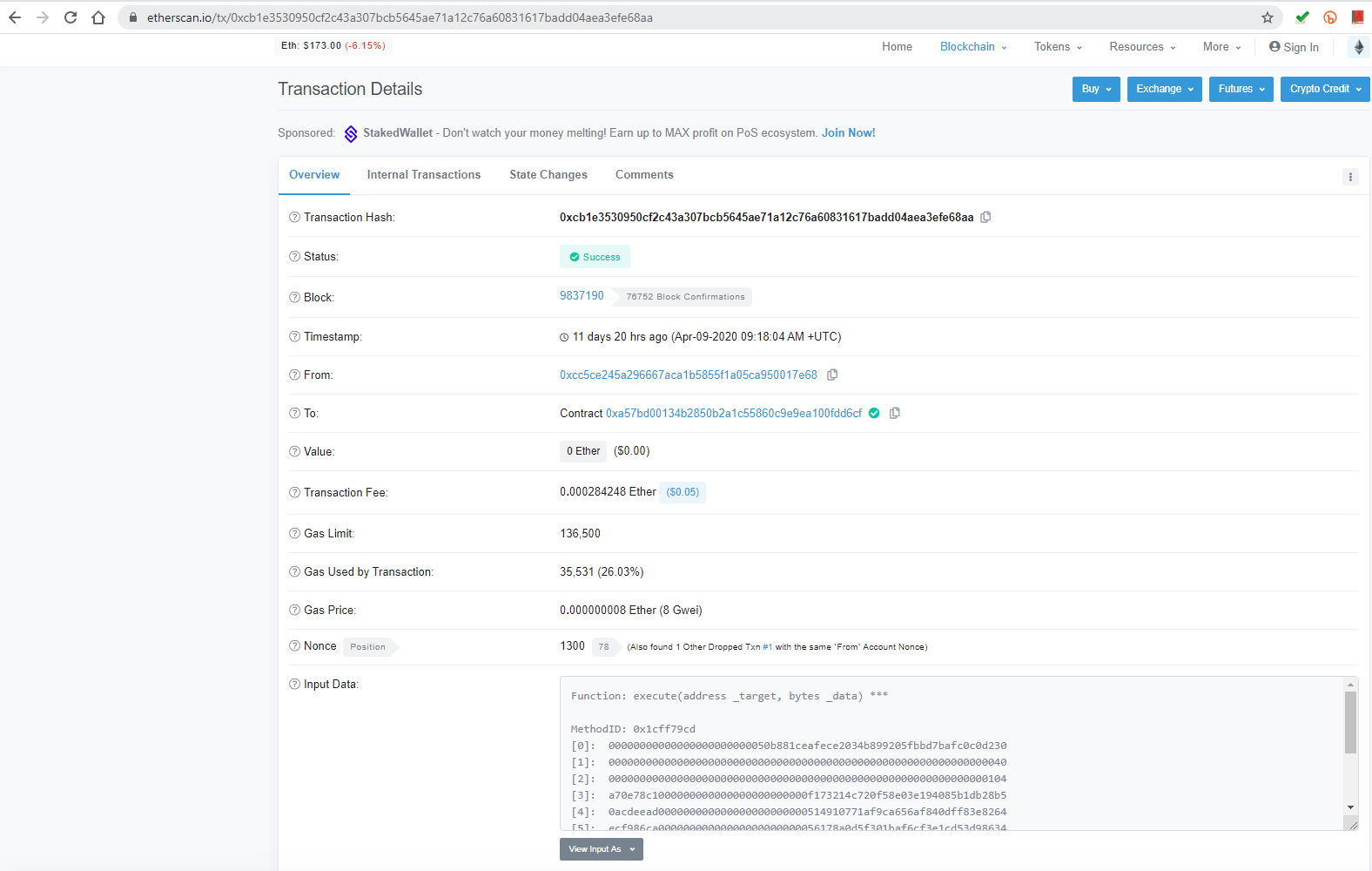
트랜잭션 처리 비용은 eth.getTransactionReceipt(hash)로 알 수 있다. 아래에서 보듯이 gasUsed는 21000이고, 이는 보통의 거래에 소요되는 gas이다. 참고로 eth.getTransactionReceipt(hash)에 전달되는 인자값은 앞에서 확인했던 블록(eth.getBlock(55169)를 실행시킨 코드)에서 transactions의 값으로 기록되었던 것이다.

> eth.getTransactionReceipt("0xd87121b8b0f84f7fa038cd7c1928ca6a222d14228125c90edc2493fdef4fb90b")  
{  
 blockHash: "0xd2c51ae5dea10e50c915e9d7ccc6c117c2d14d0f38da936b62a1c38fd0494d26",  
 blockNumber: 55169,  
 contractAddress: null,  
 cumulativeGasUsed: 21000,  
 from: "0x2e49e21e708b7d83746ec676a4afda47f1a0d693",  
 gasUsed: 21000,  
 logs: [],  
 logsBloom: "0x00000...생략...00000",  
 root: "0x1817a1775db945025d4a67a0bfcb633b5fed6a1fa76804d152db410c9140237d",  
 to: "0xe36104ad419c719e356e86f94b5a7ca47a83f9e7",  
 transactionHash: "0xd87121b8b0f84f7fa038cd7c1928ca6a222d14228125c90edc2493fdef4fb90b",  
 transactionIndex: 0  
}

### 실제

실제의 거래에 대해 gas가 얼마나 필요하고, 처리비용이 어떻게 계산되는지 살펴보자 (참조: https://etherscan.io/tx/0xcb1e3530950cf2c43a307bcb5645ae71a12c76a60831617badd04aea3efe68aa)

* 거래비용: 0.000284248 Ether ($0.05) = gas 사용량 x gas 가격 = 35531 x 8 (아래를 참조)
* Gas 한도: 136,500
* 거래의 Gas 사용량: 35,531 (26.03%)
* Gas 가격: 0.000000008 Ether (8 Gwei) 이 수준은 'Fast' 기준



alt text

# 8. 거래 건수

주소에서 전송된 거래건수는 getTransactionCount 명령어를 통해서 알 수 있다. Nonce는 0부터 계산되므로, 거래건수가 nonce보다 1만큼 크게 된다.

옵션을 주어 getTransactionCount(address, 'pending') 이런 식으로 거래건수를 구할 수도 있다.

* "latest" - 최근 블록을 의미
* "pending" - 현재 채굴된 블록 (Transaction Pool 에 남아있는 대기중 상태)

자신의 키를 입력하고, 현재 거래건수를 계산해보면 11을 출력한다.

> eth.getTransactionCount('0x8078e6bc8e02e5853d3191f9b921c5aea8d7f631')  
11

최근 블록을 살펴보자. 'transactions' 필드를 보면, 포함된 해시(hash)가 있다. 하나를 구해서 nonce 값을 확인해 보자.

geth> eth.getBlock('latest')  
{  
 difficulty: 0,  
 extraData: "0x",  
 gasLimit: 6721975,  
 gasUsed: 58533,  
 hash: "0x5a247e574a2ba56a74eb5f5d5201365563dad61eb141e333b59dfa393b50476b",  
 logsBloom: "0x00000...00000",  
 miner: "0x0000000000000000000000000000000000000000",  
 mixHash: "0x0000000000000000000000000000000000000000000000000000000000000000",  
 nonce: "0x0000000000000000",  
 number: 11,  
 parentHash: "0x36af17737106912bc85078ad63133143802d824ad36207912d267960028695d7",  
 receiptsRoot: "0xee5bc87f490e552952ea8f769e6cb2d0a84509e3a57c6975cd3c67c022cd35d9",  
 sha3Uncles: "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",  
 size: 1000,  
 stateRoot: "0xb4549f2c55629a06a09cad796a95c38db14aa58aa76cb33e34f3966a45e3e66d",  
 timestamp: 1646129233,  
 totalDifficulty: 0,  
 transactions: ["0x2a33ec94c133f92d6aea65db36805dac8131d19ae32a875006e281e128938f53"],  
 transactionsRoot: "0x3373d83bc8440c98a4fd82681546eed7240693b20fa2ae9f16083a1ea066b30c",  
 uncles: []  
}

최근 블록에 포함된 거래의 명세를 출력해 보면, nonce 값을 구할 수 있다. 어떤가? transactonCount와 비교해서 1이 작다. getTransaction()에 전달되는 값은 eth.getBlock('latest')를 실행시켜 얻은 결과에서 transactions에 있는 값이다.

geth> eth.getTransaction('0x2a33e...38f53')  
{  
 blockHash: "0x5a247e574a2ba56a74eb5f5d5201365563dad61eb141e333b59dfa393b50476b",  
 blockNumber: 11,  
 from: "0x8078e6bc8e02e5853d3191f9b921c5aea8d7f631",  
 gas: 800000,  
 gasPrice: 20000000000,  
 hash: "0x2a33ec94c133f92d6aea65db36805dac8131d19ae32a875006e281e128938f53",  
 input: "0x5f74bbde000000000000000000000000cc16c60a1fb054d2594ad06c6f8323afad7290650000000000000000000000000000000000000000000000000000000000000000",  
 nonce: 10,  
 r: "0x66e212fdd9500d61058b0c7f2dd19592ae2bf84afef7e332e2e1175980393494",  
 s: "0x33f444713f2e94e332a4b35864dc167fd7fcdde62f3672e431dacb42395caaaa",  
 to: "0x9250411625df0124335018c83521b91bde59ed5f",  
 transactionIndex: 0,  
 v: "0x25",  
 value: 0  
}

## 문제 5-1: 다른 계정으로 송금 거래

은행에서 송금 거래를 하면 계정A에서 계정B로 자금 이체가 발생한다. 동일 은행이거나 타 은행이거나 가리지 않고 송금할 수 있다.

물론 블록체인을 통해도 계정간 자금이체가 가능하다.

여기서는 동일한 기계 또는 다른 기계로의 송금을 시도해보자. 물론 송금이 가능하려면, 기계별로 계정과 지갑이 설치되어 있어야 한다.

### 1-1 같은 컴퓨터, 다른 계정으로 송금하기

동일한 컴퓨터의 타계정으로 송금을 해보자. 송금 거래를 네트워크에 전송하기 위해서는 gas와 이를 충당할 잔고가 계정에 있어야 한다.

gas가 부족하면 거래가 지연되거나 아예 처리되지 않을 수 있다. 잔고를 충전하기 위해서는 Ether를 구매하거나, 마이닝을 해서 충전을 해두어야 한다.

우선 지급 계정 account0을 선택하고, 잔고를 확인해보자. 지급계정의 송금 가능액과 gas를 충당할 잔고가있어야 한다.

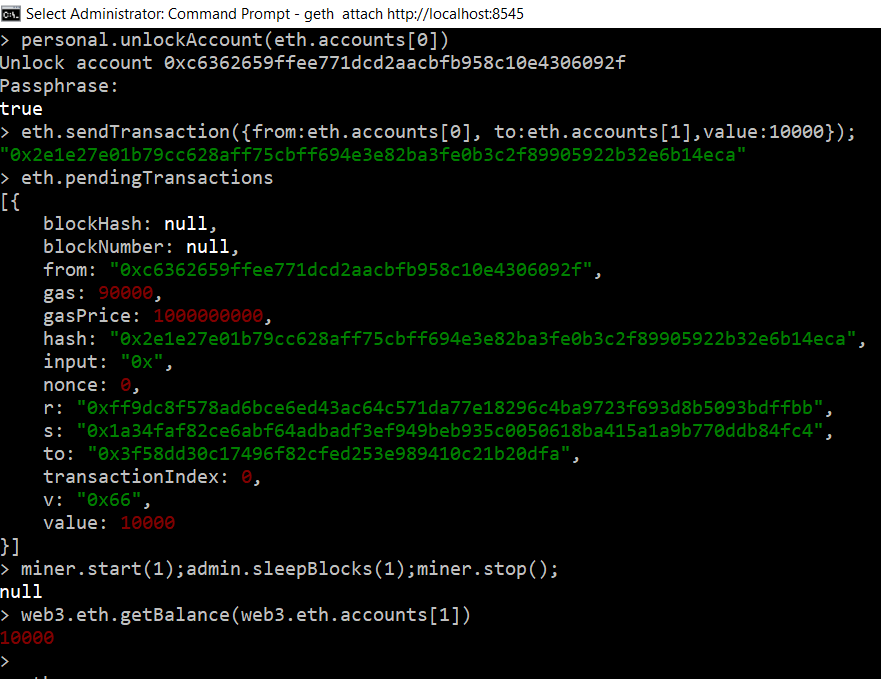
잔고가 충분히 있다는 것을 확인하고, eth.sendTransaction({from: 출금주소, to:입금주소, value:금액}) 명령어에 from: 출금주소, to: 입금주소, value: 송금액을 기입한다.

> eth.getBalance(eth.accounts[1])  
184999843394000029444  
> eth.sendTransaction({from:eth.accounts[1], to:eth.accounts[0],value:10000})  
Error: authentication needed: password or unlock  
 at web3.js:3143:20  
 at web3.js:6347:15  
 at web3.js:5081:36  
 at <anonymous>:1:1

오류가 발생하고 있는데, 이유는 계정이 잠겨있기 때문이다. 즉 거래에 필요한 gas또는 송금액이 지급되려면, 계정이 해제되어 있어야 한다. geth를 시작할 때 --unlock 하거나, console에서 직접 할 수 있다. 다음과 같이 프로그램을 실행하기 전에 다음과 같이 잠금을 해제 하자.

> personal.unlockAccount(eth.accounts[0]) 0번째 계정을 해제한다.

geth@8445에서 하게 되면, 다음 화면의 일련의 절차에 따라 실행하자: (1) 출금계정의 잠금을 해제하고, (2) 송금 거래를 블록체인에 전송하고, (3) 그 거래에 대해 마이닝한다. 그러고 나서 잔고를 확인하면서 송금이 성공적으로 이체된 것을 알 수 있다.



alt text

물론 geth 단말에서 한 줄씩 실행해도 좋지만, 여기서는 프로그램을 만들어 일괄 실행해보자.

[프로그램: src/e\_testTran.js]  
줄01 miner.setEtherbase(eth.accounts[0]);  
줄02 console.log('coinbase: ', eth.coinbase);  
줄03 var bal1=eth.getBalance(eth.coinbase);  
줄04 var bal2=eth.getBalance(eth.accounts[1]);  
줄05 console.log('sender balance in ether: ', web3.fromWei(bal1,"ether"));  
줄06 console.log('receiver balance in ether: ', web3.fromWei(bal2,"ether"));  
줄07 console.log('median gas price: ', eth.gasPrice);  
줄08 console.log('block number: ', eth.blockNumber);  
줄09 console.log('transaction count: ', eth.getTransactionCount(eth.coinbase));  
줄10 eth.sendTransaction({from:eth.coinbase, to:eth.accounts[1],value:10000});  
줄11 console.log('...mining start');  
줄12 miner.start(1);admin.sleepBlocks(1);miner.stop();  
줄13 console.log('mining done...');  
줄14 var bal1new=eth.getBalance(eth.coinbase);  
줄15 var bal2new=eth.getBalance(eth.accounts[1]);  
줄16 console.log('- new sender balance in ether: ', web3.fromWei(bal1new,"ether"));  
줄17 console.log('- new receiver balance in ether: ', web3.fromWei(bal2new,"ether"));  
줄18 console.log('- block number: ', eth.blockNumber);  
줄19 console.log('- transaction count: ', eth.getTransactionCount(eth.coinbase));

* 줄1 지급계좌 coinbase를 설정한다.
* 줄7 eth.gasPrice() 지난 몇 건 블록의 중위값을 알려준다.
* 줄8, 줄18 블록번호를 출력한다. 사적망이라 거래 하나에 블록이 하나 생성되고 있다.
* 줄9, 줄19 거래건수가 하나 증가한다.
* 줄12 사적망이라서 자신이 거래를 발생하여 스스로 마이닝을 하자. 마이닝하기 전 후 txpool에 대기로 잡히다가 소진되는지 살펴보자.

파일을 저장하고 나서, 일괄실행을 해보자. 송금거래가 발생하고, 마이닝을 하고, 잔고와 block number가 변화하게 된다. coinbase에는 마이닝 보상까지 더해지므로, 잔고가 송금액만큼 줄어들지는 않을 것이다.

geth> geth --exec "loadScript('src/e\_testTran.js')" attach http://localhost:8445  
  
coinbase: 0x21c704354d07f804bab01894e8b4eb4e0eba7451  
sender balance in ether: 60.000021786999979076  
receiver balance in ether: 184.999843394000019444  
median gas price: 1000000000 #사적망에서는 가격이 정해져 있다.  
block number: 47206  
tranaction count: 264  
...mining start  
mining done...  
- new sender balance in ether: 65.000021786999969076 #10000 wei 출금 후 마이닝보상 증가  
- new receiver balance in ether: 184.999843394000029444 #10000 wei 입금액만큼 증가한다.  
- block number: 47207 #블록이 1 증가.  
- tranaction count: 265 #거래건수 1증가.  
true

### 1-2 다른 기계, 다른 계정으로 송금하기

은행이라고 생각하면, 타행 이체라고 간주할 수 있는 거래이다.

블록체인으로 보면, 노드A에서 노드B ("0x519775cc61e4c9b3f19b75426a7a3696a3c85035") 으로 송금하는 사례이다.

노드B의 계정은 따로 복사해서 붙여넣기 하고 있다. 실행하고 나면, 잔고가 10000 송금액만큼 증가하고 있다. 노드A와 노드B는 동일한 사설망의 멀티노드들이다. 앞서 노드A에 노드B가 Peer로 추가되어 있다. 혹시 추가가 되어 있지 않다면 송금이 성공하지 못하게 될 것이다.

[프로그램: src/e\_testTran2.js]  
var hqaccount="0x519775cc61e4c9b3f19b75426a7a3696a3c85035";  
console.log('NodeB account balance: ', eth.getBalance(hqaccount));  
console.log('block number: ', eth.blockNumber);  
var t=eth.sendTransaction({from:eth.accounts[0], to:hqaccount, value:10000});  
console.log('transactionHash: ',t);  
console.log('...mining start');  
miner.start(1);admin.sleepBlocks(1);miner.stop();  
console.log('mining done...');  
console.log('block number: ', eth.blockNumber);  
console.log('NodeB account balance: ', eth.getBalance(hqaccount));

노드A에서 프로그램을 일괄 실행해보자.

pjt\_dir> geth --exec "loadScript('src/e\_testTran2.js')" attach http://localhost:8445  
  
NodeB account balance: 20000  
block number: 70296  
transactionHash: 0x15b17f04f20322cc8ee14aef9a9b617ec77935f29ce313a78ad1388241607e6c  
...mining start  
mining done...  
block number: 70297  
NodeB account balance: 30000 #10000 wei 증가하고 있다.

## 연습문제

1. 블록체인에 읽기를 하거나 기록을 하는 경우로 함수를 구분할 수 있다. 어떤 함수가 사용되는지 명칭을 적으시오.
2. ABI는 함수호출을 머신코드 수준으로 표현한 것이다. OX로 답하시오.
3. ABI를 적용하면, 함수 선택자는 ( )바이트, 매개변수는 ( )바이트로 이루어진다.
4. 이더리움 트랜잭션의 필수 항목은 무엇인지 적으시오.
5. 이더리움 트랜잭션의 항목이 아닌 것을 고르시오.

* from
* to
* value
* gas
* gasPrce
* data
* nonce
* balance

1. 이더리움과 비트코인의 거래를 구성하는 항목에 차이가 있다. OX로 답하시오.
2. 블록에 포함되지 않는 거래는 gas를 2배 지급하지 않으면 버려진다. OX로 답하시오.
3. 블록은 전 블록의 머클 루트를 통해 체인으로 연결된다. OX로 답하시오.
4. 블록 헤더에 포함되지 않는 해시를 고르시오.

* 거래 트리의 해시
* 거래 수령 트리의 해시
* 저장 트리의 해시
* 상태 트리의 해시

1. 블록헤더에는 gasLimit, gasUsed 항목이 포함된다. OX로 답하시오.
2. 최근 블록의 명세를 출력해서, 포함된 항목을 적으시오.
3. 머클 증명을 통해 어떤 거래도 수정될 수 없다는 의미를 설명하시오.
4. gas 단가는 너무 적으면 마이닝이 되지 않을 수 있고, 많으면 빠르게 될 수 있다. OX로 답하시오.
5. 거래가 6개 있고 gas비가 각 25, 30, 35, 40, 45, 50이라고 하자. 블록의 거래한도는 100이라고 하자. 그렇다면 블록에는 어떤 거래가 포함될 수 있는지 설명하시오.
6. 블록체인에 배포할 경우 컨트랙을 생성해야 하는데, 기본 gas가 얼마나 필요한지 적으시오.
7. Nonce는 0부터 계산되므로, 거래건수가 nonce보다 1만큼 크게 된다. OX로 답하시오.
8. 송금 거래에서 발생하는 gas는 출금 계정, 입금 계정 1/2씩 부담한다. OX로 답하시오.
9. 블록이 너무 빨리 또는 느리게 만들어지는 것을 막기 위해 무엇을 조정하는가?
10. gas
11. difficulty
12. nonce
13. gas는 데이터의 성격과 송금 금액에 따라 결정된다. OX로 답하시오.
14. 다음 네 건의 데이터에 대해 머클 루트(Merkle Root) 값을 계산하고 출력하시오. 중간 노드의 AB, CD의 해시도 출력하세요.

* txA = 'Hello'
* txB = 'How are you?'
* txC = 'This is Thursday'
* txD = 'Happy new Year'

1. 친구의 주소를 구하고, 그 주소로 송금해 보자. 송금이 되지 않으면 왜 안되는지 이유를 알아보자.

* 송금 전후, 자신과 친구 계정잔고의 증가분을 출력하시오.
* 소요된 gas비용 출력하세요.

1. 다음 sayHello()함수의 ABI 명세를 생성한다.

contract Hello {  
 function sayHello(bytes toWhom) pure public returns(string memory) {}  
}

1. "Let's meet in my office at 10 AM."의 거래비용 gas를 계산하시오.
2. **목표 해시값(Target Hash)을 100미만의 양수**로 정해진다고 하고, 이를 마이닝하는 프로그램을 작성하시오. 해시는 별도로 SHA 해싱할 필요없이 십진수로 생각하자. **NONCE는 반복회수**로만 쓰이고 무작위 수를 생성하는데 입력되지는 않는다고 가정하자.

목표 해시값을 90으로 또는 10으로 정해서 **몇 회 만에 그 값을 찾는지 비교**해 보자. 여기서 난이도를 3회 만에 찾게 되면 1이라고 하자. 3회 보다 횟수가 많이 걸리면 이 경우는 어렵다는 의미이므로 난이도를 낮추어야 한다. 반대의 경우 3회 보다 적은 횟수가 필요하면 난이도를 높이게 된다.

* 90을 목표해시로 정하고 몇 회만에 마이닝에 성공하는지 출력하고, 난이도를 평가하시오.
* 10을 목표해시로 정하고 몇 회만에 마이닝에 성공하는지 출력하고, 난이도를 평가하시오.
* 참조: random 함수 random()은 0 ~ 1사이의 무작위 수를 생성한다. 이 함수에 100을 곱하면 0 ~ 100 사이의 수를 생성한다 (100은 제외). print문의 end는 출력을 이어서 하게 만든다.

from random import randint  
for i in range(1,20):  
 print(int(random.random()\*100), end=" ")  
83 18 99 61 2 74 4 51 12 63 46 64 67 72 7 70 73 49 86

1. 블록헤더 데이터의 해시 값에 NONCE를 증가시키면서 앞 자리의 0의 개수를 맞출 때까지 반복한다.
2. 찾고자 하는 해시가 0000로 시작한다고 하자. 몇 회만에 찾는지 출력
3. 찾고자 하는 해시가 00000로 시작한다고 하자. 몇 회만에 찾는지 출력
4. 찾고자 하는 해시가 000000로 시작한다고 하자. 몇 회만에 찾는지 출력

난이도가 어떤 경우가 높았으며, 난이도에 따라 찾는 회수의 차이가 있는지 설명하시오.

1. geth서버를 포트번호 8446에 하나 더 띄우시오 (geth@8446이라고 명명). geth@8446의 chainid, nteworkid는 36번으로 설정한다. geth@8446에서 계정을 2개 만들고, 충전을 해 놓는다 (coinbase에 5 Ether 이상). 아래 문제는 eth 스크립트로 작성하여 풀고, geth --exec 'loadScript()'로 실행한다.

* 1-1 geth@8446에서 admin.nodeInfo 출력
* 1-2 geth@8446에서 계정, ether 잔고 출력 (잔고가 5 ether 이상 있어야 함)
* 1-3 geth@8446에서 블록번호를 출력
* 1-3 geth@8446 coinbase에서 geth@8446 2번째 계정으로 1.1111 ether 계좌이체
* 1-4 계좌이체의 hash값을 사용해 getTransactionReceipt 출력
* 1-5 계좌이체가 성공했다면 geth@8446의 수신측 계정잔고, 수신측 잔고변화 ether, 블록번호를 출력

1. geth서버를 포트번호 8446에 하나 더 띄우시오 (geth@8446이라고 명명) 현재 8445 포트를 쓰고 있는 geth(geth@8445라고 명명)를 포함해서, 총 2개의 geth 서버가 실행된다. geth가 2개가 뜨지 않으면, geth 띄울 때 --ipcdisable 스위치를 추가한다. 아래 문제는 eth 스크립트로 작성하여 풀고, geth --exec 'loadScript()'로 실행한다.

* 1-1 geth@8446 coinbase에서 geth@8445으로 coinbase로 1.11 ether 계좌이체
* 1-2 계좌이체의 hash값을 사용해 getTransactionReceipt 출력
* 1-3 계좌이체가 성공했는지, 실패했는지 적으시오. 성공했다면 수신측 geth@8445의 계정, 잔고 ether, 블록번호를 출력하고 실패했다면 그 이유를 적으시오.

1. geth서버를 포트번호 8446에 하나 더 띄우시오 (이하 geth@8446이라고 함). geth@8446의 nteworkid는 36번으로 설정한다. geth@8446에서 계정을 2개 만들고 (이하 계정A, 계정B라고 함), 충전을 해 놓는다 아래 문제는 eth 스크립트로 작성하여 풀고, geth --exec 'loadScript()'로 실행한다. 스크립트 파일은 하나 이상 작성해도 된다. 아래 문제는 모두 geth@8446에서 실행한다. 참고로 JSON 형식이라서, "Object" 내용을 볼 수 없는 경우 JSON.stringify()함수를 사용하면 문자열로 변환되어 출력된다.

* 1-1 admin.nodeInfo를 살펴보면, ip와 chainId가 있다. 이 두 정보를 출력하자. 부가적으로 if문을 사용하여 자신이 설정한 chainId 36이 private network인지 아닌지 출력한다.
* 1-2 계정A, 계정B의 wei, ether 잔고를 출력한다 (5 ether 이상). 그리고 transaction count를 출력한다.
* 1-3 계정A -> 계정B로 0.00000000000010101 ether를 계좌이체하고, 해시를 구한다. 이 거래는 마이닝이 필요하다. 마이닝을 시작할 때, 해시값과 더불어 시작한다는 출력을 한다. 끝나면 끝나면 "mining done" 출력하시오.
* 1-4 계좌이체 이후, 잔고와 transaction count를 출력한다. 차이는 계산하여 출력한다. 계좌이체의 hash값을 사용해 getTransactionReceipt의 gasUsed를 출력한다. gasUsed를 한화로 출력한다. 1 ether에 2,500,000원으로 가정한다.