

HYPERLEDGER FABRIC v1.0

운영자 가이드

HYPERLEDGER FABRIC 네트워크 구성
MOO JE KONG

1. Hyperledger Fabric v1.0 Overview	3
1.1. 시스템 아키텍처.....	3
1.1.1. 트랜잭션.....	4
1.1.2. 블록체인 데이터 구조.....	4
1.1.3. 노드.....	5
1.1.4. 트랜잭션의 흐름.....	5
2. Hyperledger Fabric 기반의 블록체인 네트워크 구성하기	7
2.1. 사전 준비 사항	7
2.2. Hyperledger Fabric 소스 받기 및 빌드 방법.....	7
2.3. Fabric 네트워크 부트스트랩을 위한 준비.....	8
2.3.1. <i>Crypto Generator</i>	8
2.3.2. <i>Configuration Transaction Generator</i>	8
2.3.3. 인증서 및 아티팩트 생성.....	9
3. Fabric 네트워크 시작	9
3.1. 채널 생성	10
3.2. 채널에 피어 참여	12
3.2.1. <i>Peer0.org1</i>	12
3.2.2. <i>Peer1.org1</i>	12
3.2.3. <i>Peer0.org2</i>	12
3.2.4. <i>Peer1.org2</i>	13
3.3. Anchor 피어 설정	13
3.4. 체인코드 디플로이 및 동작 확인.....	13
3.4.1. 체인코드 디플로이.....	14
3.4.2. 체인코드 초기화.....	14
3.4.3. <i>Query</i>	15
3.4.4. <i>Invoke</i>	15
3.4.5. <i>Query</i>	15

1. Hyperledger Fabric v1.0 Overview

Hyperledger Fabric 은 2017 년 3 월 16 일 첫 번째 알파 버전이 릴리스되면서 프로젝트가 인큐베이션(Incubation)에서 액티브(Active) 상태로 바뀌었고 이 문서를 작성하는 2018 년 3 월 현재는 v1.1.0-alpha 버전이다.

Hyperledger Fabric v1.0 에서는 많은 부분 아키텍처의 변화가 있고 그에 따라 구성 컴포넌트에도 다수의 변화가 있었다. 새로운 아키텍처는 높은 기밀성, 복원성, 유연성 및 확장성을 지원하기 위한 모듈러(modular) 방식을 더욱 충실히 지원함으로써 현실에서의 복잡한 비즈니스 환경을 수용하도록 설계됐다.

1.1. 시스템 아키텍처

Fabric v1.0 의 아키텍처에서 가장 큰 변화는 피어의 역할이다. 이전 버전(Fabric v0.6)까지는 Validating Peer 에서 합의 알고리즘과 원장 관리 등 블록체인이 수행해야 할 모든 동작을 담당하고 있었으나 새로운 아키텍처에서는 Endorsing Peer, Committing Peer, Ordering Node 에서 나눠서 동작을 역할에 맞게 수행하게 되며, 멤버십 서비스는 고가용성을 위해 클러스터링 구성이 가능해졌다. 이러한 모든 노드에 대해 트랜잭션 흐름이 이전 버전에서는 클라이언트 애플리케이션에서 한번의 트랜잭션 제출로서 완료되었으나, 새로운 아키텍처에서는 클라이언트와 노드들이 여러 단계를 거쳐 통신을 주고 받아 하나의 트랜잭션을 완료하도록 변경되었다. 클라이언트 애플리케이션 단에서 트랜잭션을 처리하는 단위는 Hyperledger Fabric SDK 이며 트랜잭션을 처리하는 동작에 대한 비중이 이전 버전에 비해 더욱 커졌다.

변화된 Hyperledger Fabric v1.0 아키텍처에서 가장 큰 특징은 트랜잭션을 처리할 때 트랜잭션을 검증하고 보증하기 위한 노드가 추가됨으로써 비확정적인 트랜잭션을 제거할 수 있게 됐고, 채널이라는 개념을 통해 비즈니스적으로 차별성과 확장성이 더해졌다는 것이다.

관리적인 측면에서는 원장을 관리하는 스테이트 DB 를 변경할 수 있어서 이전 버전에 비해 DB 특유의 다양한 쿼리를 사용해 원장에 대해서 풍부한 쿼리를 제공할 수 있다는 장점이 있다.

합의 과정을 관리하는 Orderer 노드는 "Solo", "Kafka", "SBFT(향후 추가 예정)" 등을 선택할 수 있으며 Fabric 네트워크를 구성하는 모든 피어들에게 동일한 원장을 저장할 수 있게 하는 역할을 담당한다.

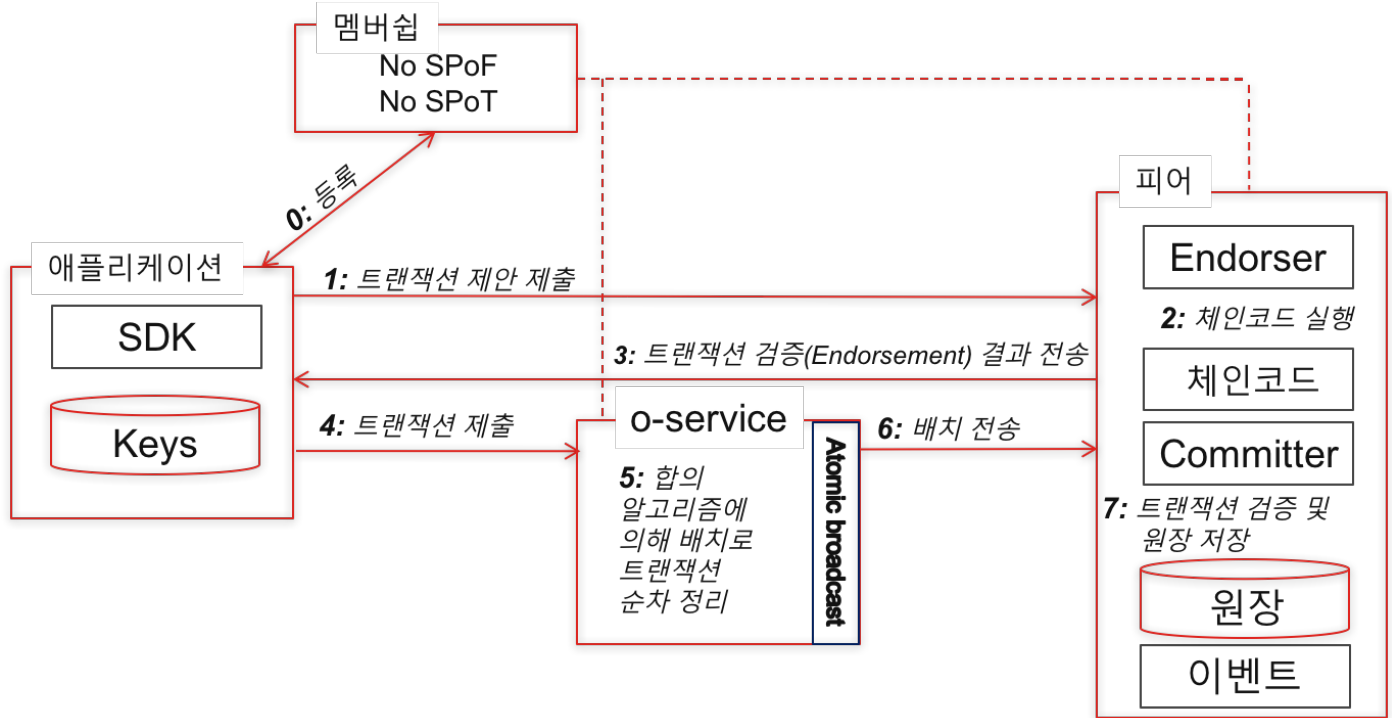


그림 1-1 Hyperledger Fabric v1.0 아키텍처

1.1.1. 트랜잭션

Hyperledger Fabric 에서 트랜잭션은 크게 두 가지로 나뉜다.

1.1.1.1. 배포 트랜잭션(Deploy transaction)

배포 트랜잭션은 단어 그대로 체인코드를 블록체인의 시스템에 설치하기 위해 실행되며, 배포 트랜잭션이 호출되면 체인코드를 빌드하고 피어에 설치하며 초기값을 파라미터로 받아서 블록에 저장하는 과정을 거친다.

1.1.1.2. Invoke 트랜잭션

배포된 체인코드를 기준으로 정의된 함수를 호출하기 위해 실행된다. Invoke 트랜잭션을 호출한 결과, 호출이 정상적으로 완료되면 스테이트에 처리 결과가 저장되고 결과는 호출한 클라이언트에 반환된다.

1.1.2. 블록체인 데이터 구조

1.1.2.1. 스테이트(State)

블록체인에서 최신 State 는 버전 관리된 키-밸류 스토어(KVS)로 설계돼 있다. KVS 에 문자열 타입의 키와 임의의 객체인 밸류로 저장 가능하며, 블록체인에서 구동되고 있는 체인코드에서 "PUT", "GET" 동작에 의해 관리된다. 키-밸류 스토어는 동시에 버전 정보를 함께 저장하고 있어서 동일한 키 값으로 입력되는 밸류에 대해 버전 관리가 이뤄진다. 그러므로 모든 거래되는 데이터에 대해서는 히스토리 관리가 된다.

1.1.2.2. 원장(Ledger)

원장은 블록체인에서 모든 성공 또는 실패한 스테이트의 변경에 대한 검증 가능한 기록을 저장한다. 원장은 Ordering 서비스에 의해 완벽하게 순차 처리된 트랜잭션의 해시 체인으로 구성되며, 해시 체인은 원장에 블록의 전체 순서를 지정하며 각 블록에는 순차로 정렬된 트랜잭션 배열이 포함된다.

원장은 모든 피어와 일부 Orderer 에서 저장되고 관리된다. 피어에는 원장과 스테이트가 관리되는데, 역할에 따라서 Committing Peer 와 Endorsing Peer 로 구분되며 Endorsing Peer 에 의해 검증된 트랜잭션뿐 아니라 검증에 실패한

트랜잭션의 정보도 비트 구분을 통해서 함께 관리한다. Orderer 는 결함 허용정보(Fault-tolerance) 와 피어의 가용성 정보를 원장으로 관리한다.

1.1.3. 노드

블록체인에서 노드는 통신을 위한 엔티티를 말하며, 같은 유형의 여러 노드가 동일한 서버에서 구동될 수 있다는 점에서 논리적인 기능으로 구분되는 단위다. 노드는 크게 세 가지로 구분된다.

1. **클라이언트**: 트랜잭션을 발생시켜 Endorser 에게 트랜잭션을 제안(Proposal)하고 Endorsing 결과를 받아서 최종적으로 Orderer 에게 트랜잭션을 전송하는 역할을 하는 노드
2. **피어**: 트랜잭션을 커밋하고 원장을 관리가 주된 역할이며, 설정에 의해 Endorser 의 역할을 할 수도 있다.
3. **Ordering 서비스 노드(또는 Orderer)**: 트랜잭션에 대해 각 피어에게 일관되게 동일한 원장을 관리할 수 있도록 커뮤니케이션을 보장하는 역할을 하는 노드

1.1.4. 트랜잭션의 흐름

Hyperledger Fabric v1.0 에서는 하나의 거래가 완료되기까지 어떤 과정으로 처리되는지 간략하게 알아보려고 한다. 이전 버전(Fabric v0.6)에서는 "Invoke"라는 한 번의 동작으로 완료됐으나 새로운 아키텍처에서는 다소 복잡해 보이기도 한다. 이 과정은 비확정적인 트랜잭션을 사전에 제거하는 결과를 가져온다. 결국 이 과정은 Fabric 네트워크와 SDK 를 통해 이뤄지며 SDK 가 이전 버전에 비해서 많은 역할을 하게 됐다.

1.1.4.1. 클라이언트로 부터 트랜잭션 제안

최초 클라이언트가 거래를 일으키게 되면 클라이언트의 SDK 는 트랜잭션에 대한 제안을 Endorsing 피어들에게 전송한다. 이때 전송되는 정보는 주로 클라이언트의 인증서와 트랜잭션 데이터다. 여기서 클라이언트가 트랜잭션을 제안해야 할 상대 Endorsing 피어는 Endorsement 정책에 의해 설정된 피어들이다. 예를 들어, 채널에 PeerA, PeerB 가 있으며 Endorsement 정책에 의해 모든 피어들로부터 트랜잭션을 검증 받아야 한다고 설정돼 있다면 PeerA, PeerB 에게 트랜잭션을 제안하게 된다.

1.1.4.2. Endorser 에 의한 트랜잭션 검증

트랜잭션의 제안은 Endorsing 피어가 받게 되는데 Endorsing 피어들은 1) 제안의 내용이 적절한 포맷으로 이뤄져 있는지, 2) 이전에 제안된 트랜잭션이 아닌지, 3) 제안자의 서명(인증서)이 유효한지, 4) 제안자가 채널에 대해 트랜잭션을 시작할 수 있는 자격이 있는지 등을 검증하고 최종적으로 제안 내용으로 체인코드를 가상으로 실행해 정상적으로 실행되는 제안인지 검증한다.

최종적으로 검증이 완료되면 검증의 결과를 Endorsing 피어의 서명과 YES/No 의 데이터 등을 포함해서 클라이언트, 즉 클라이언트의 SDK 에게 전송한다.

1.1.4.3. 제안 검증 결과 검토

Endorsing 피어들이 제안 검증 결과를 클라이언트에 반환하게 되면 클라이언트는 검증 결과를 검토해서 Endorsement 정책에 부합되는지 등을 검토한다. 이 모든 동작은 SDK 에 의해 일어난다.

1.1.4.4. 트랜잭션 제출

클라이언트에서 최종적으로 검토가 완료되어 트랜잭션의 결과를 원장으로 저장해도 되는 상태가 되면 Ordering 서비스에 트랜잭션을 제출한다. 제출할 때는 "Broadcast" 방식으로 Ordering 서비스에 제출하며, Ordering 서비스에 연결된 모든 채널로부터 검토된 트랜잭션이 배치 형태로 제출된다.

이때 Ordering 서비스는 이미 내용이 검토된 트랜잭션이므로 내용 검토를 하지 않고 채널별로 제출된 트랜잭션을 순서에 맞춰 정렬하며 피어에 저장하기 위한 블록을 생성한다.

1.1.4.5. 트랜잭션 커밋

Ordering 서비스에 의해 생성된 새로운 블록은 해당하는 채널의 피어들로 “배달”된다. 새로운 블록을 받은 피어들은 다시 Endorsement 정책에 위반되지 않는지 검증하며 최종적인 결과를 블록에 태깅한다.

1.1.4.6. 원장에 저장

모든 과정이 완료된 블록은 채널이 관리하는 체인의 뒷부분에 첨부되어 하나의 체인으로 관리된다. 최종 처리 결과는 클라이언트에게 이벤트를 통해 알려진다.

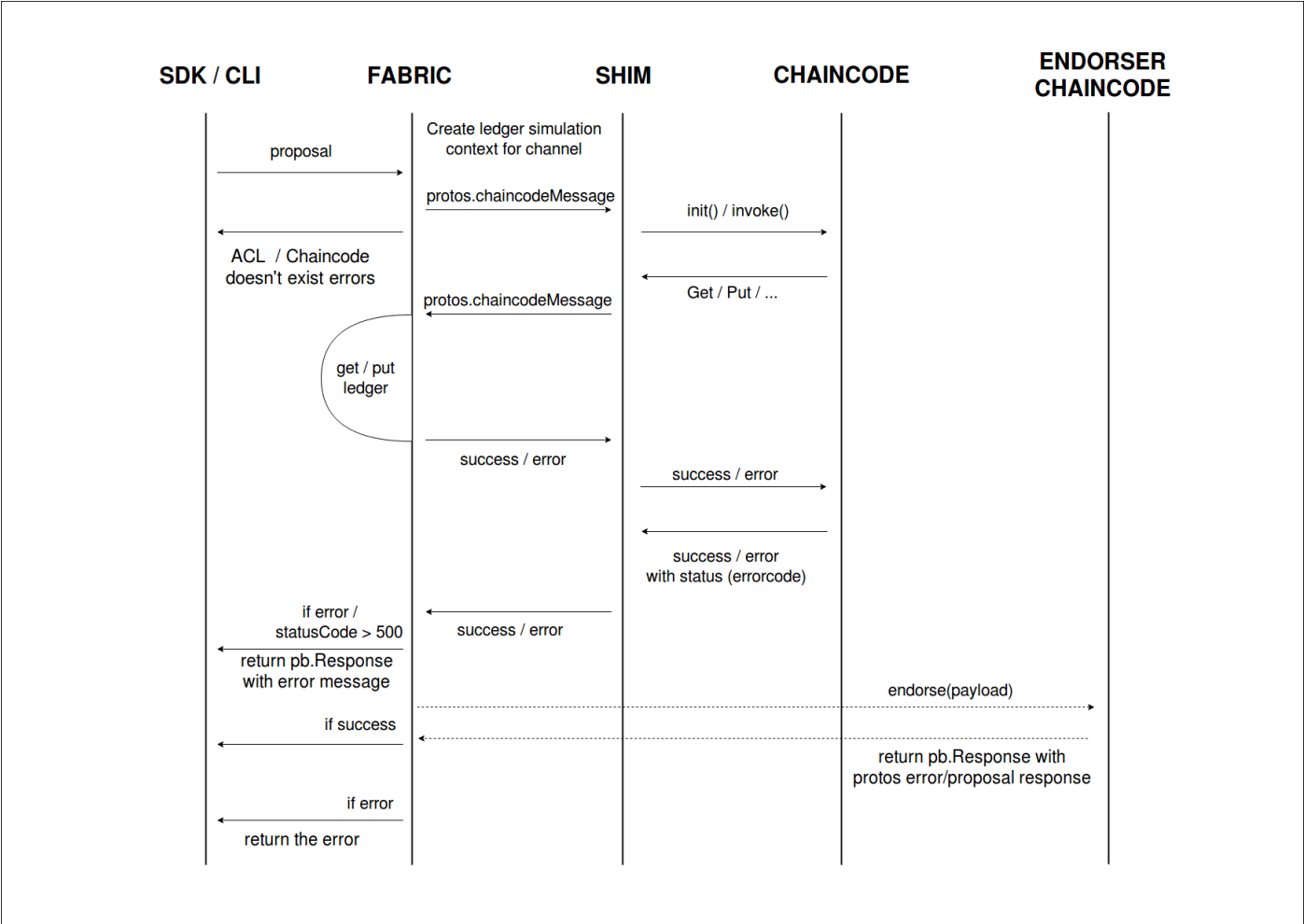


그림 1-2 트랜잭션 흐름 [출처 : <http://hyperledger-fabric.readthedocs.io/en/latest/chaincode.html#swimlane>]

2. Hyperledger Fabric 기반의 블록체인 네트워크 구성하기

이번 절에서는 Hyperledger Fabric v1.0 을 기준으로 블록체인 네트워크가 어떻게 동작하는지 간단하게 알아보기 위해 샘플 Fabric 네트워크를 구성해본다. Ordering 서비스는 'Solo' 로 구성하며, 두 개의 참여기관이 두 개의 피어로 각각 구성된다고 가정한다.

- 이 글을 쓰는 현재 최신 버전인 v1.0.0-rc1 버전을 기준으로 설명한다.

2.1. 사전 준비 사항

샘플 Fabric 네트워크 환경은 Ubuntu 16.04 환경을 기준으로 테스트했으며, 다음 항목을 미리 설치해서 진행한다(패키지별 자세한 설치 방법은 각 사이트에서 확인할 필요가 있다).

- Docker: 17.06.2-ce 이상
- Docker-compose: v1.14 이상
- Golang: 1.9.x 이상
- Node.js: 8.9.x 이상(Node.js 9.x 버전은 이 문서가 작성되는 시점에서는 지원되지 않음)
- Npm: 5.6.x 이상

2.2. Hyperledger Fabric 소스 받기 및 빌드 방법

Golang 을 설치한 후 GOPATH 환경변수를 설정했으면 다음 명령을 통해 Hyperledger Fabric 소스를 다운받는다.

```
go get github.com/hyperledger/fabric
go get github.com/hyperledger/fabric-ca
```

소스 받기가 완료되면 다음 위치 아래의 fabric, fabric-ca 디렉터리에 소스가 받아졌는지 확인한다.

```
cd $GOPATH/src/github.com/hyperledger/
```

다음 단계로 빌드를 진행한다.

```
## Fabric 소스를 빌드한다.
cd $GOPATH/src/github.com/hyperledger/fabric
git checkout -b v1.1.0
make native docker

## Fabric 소스 빌드 완료 후 fabric-ca 도 빌드한다.
cd $GOPATH/src/github.com/hyperledger/fabric-ca
git checkout -b v1.1.0
make docker
```

빌드가 완료되면 다음과 같은 결과를 볼 수 있으며, `docker images` 명령을 통해 빌드된 이미지를 확인한다.

```
mjkong@bcdev:~/go/src/github.com/hyperledger/fabric-ca$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hyperledger/fabric-ca-tools	latest	0b7fe90c12bd	5 minutes ago	1.53GB
hyperledger/fabric-ca-tools	x86_64-1.1.0-beta-snapshot-680960e	0b7fe90c12bd	5 minutes ago	1.53GB
hyperledger/fabric-ca-peer	latest	122f11d98235	5 minutes ago	253MB
hyperledger/fabric-ca-peer	x86_64-1.1.0-beta-snapshot-680960e	122f11d98235	5 minutes ago	253MB
hyperledger/fabric-ca-orderer	latest	133c7b433381	7 minutes ago	247MB
hyperledger/fabric-ca-orderer	x86_64-1.1.0-beta-snapshot-680960e	133c7b433381	7 minutes ago	247MB
hyperledger/fabric-ca	latest	cabf75e5a9f0	8 minutes ago	283MB
hyperledger/fabric-ca	x86_64-1.1.0-beta-snapshot-680960e	cabf75e5a9f0	8 minutes ago	283MB
hyperledger/fabric-tools	latest	22c43d3c607d	11 minutes ago	1.46GB
hyperledger/fabric-tools	x86_64-1.1.0-beta-snapshot-0a81773	22c43d3c607d	11 minutes ago	1.46GB
hyperledger/fabric-testenv	latest	af6f9e5a924e	12 minutes ago	1.56GB
hyperledger/fabric-testenv	x86_64-1.1.0-beta-snapshot-0a81773	af6f9e5a924e	12 minutes ago	1.56GB
hyperledger/fabric-buildenv	latest	ca3b128d5a99	13 minutes ago	1.45GB
hyperledger/fabric-buildenv	x86_64-1.1.0-beta-snapshot-0a81773	ca3b128d5a99	13 minutes ago	1.45GB
hyperledger/fabric-orderer	latest	a64e19fae0f3	13 minutes ago	180MB
hyperledger/fabric-orderer	x86_64-1.1.0-beta-snapshot-0a81773	a64e19fae0f3	13 minutes ago	180MB
hyperledger/fabric-peer	latest	2dc2edc4b63d	13 minutes ago	187MB
hyperledger/fabric-peer	x86_64-1.1.0-beta-snapshot-0a81773	2dc2edc4b63d	13 minutes ago	187MB
hyperledger/fabric-javaenv	latest	9ece7b4067a2	15 minutes ago	1.52GB
hyperledger/fabric-javaenv	x86_64-1.1.0-beta-snapshot-0a81773	9ece7b4067a2	15 minutes ago	1.52GB
hyperledger/fabric-ccenv	latest	80859202739a	16 minutes ago	1.39GB
hyperledger/fabric-ccenv	x86_64-1.1.0-beta-snapshot-0a81773	80859202739a	16 minutes ago	1.39GB
hyperledger/fabric-baseimage	x86_64-0.4.6	dbe6787b5747	10 days ago	1.37GB
hyperledger/fabric-zookeeper	latest	92cbb952b6f8	10 days ago	1.39GB
hyperledger/fabric-zookeeper	x86_64-0.4.6	92cbb952b6f8	10 days ago	1.39GB
hyperledger/fabric-kafka	latest	554c591b86a8	10 days ago	1.4GB
hyperledger/fabric-kafka	x86_64-0.4.6	554c591b86a8	10 days ago	1.4GB
hyperledger/fabric-couchdb	latest	7e73c828fc5b	10 days ago	1.56GB
hyperledger/fabric-couchdb	x86_64-0.4.6	7e73c828fc5b	10 days ago	1.56GB
hyperledger/fabric-baseos	x86_64-0.4.6	220e5cf3fb7f	10 days ago	151MB
hyperledger/fabric-tools	x86_64-1.1.0-alpha	b7b3d30505d4	4 weeks ago	1.44GB
hyperledger/fabric-orderer	x86_64-1.1.0-alpha	1750a681d781	4 weeks ago	163MB
hyperledger/fabric-peer	x86_64-1.1.0-alpha	f00c5d490d19	4 weeks ago	170MB

그림 2-1 Hyperledger Fabric 소스로 빌드된 도커 이미지 리스트

2.3. Fabric 네트워크 부트스트랩을 위한 준비

Hyperledger Fabric 을 이용하여 블록체인 네트워크 생성을 위한 실습을 하기 위해서 다음과 같은 방법으로 샘플을 다운로드 받습니다.

```
cd $HOME
curl -sSL https://goo.gl/6wtTN5 | bash -s 1.1.0
```

Fabric v1.0 부터는 블록체인 네트워크를 구성하기 위해서는 다양한 PKI 기반의 인증서와 아티팩트가 필요하다. 이를 쉽게 생성하고 관리하기 위해 툴을 제공하고 있으며, Crypto Generator(cryptogen), Configuration Transaction Generator(configtxgen)이 여기에 해당한다.

2.3.1. Crypto Generator

블록체인 네트워크를 구성하는 다양한 엔티티에서 사용하게 될 x509 기반 인증서를 생성한다.

crypto-config.yaml 파일에 필요한 네트워크 토폴로지를 작성해서 툴을 실행하면 Organization 별로 루트 인증서(ca-cert), 개인키(keystore), 공개키(signcerts) 가 생성된다.

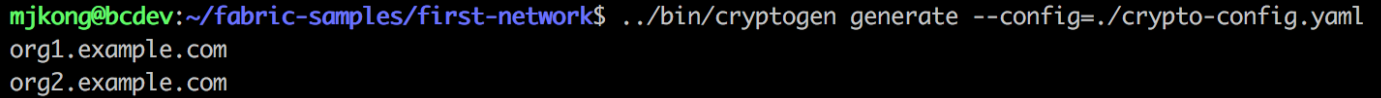
2.3.2. Configuration Transaction Generator

configtxgen 은 블록체인 네트워크를 시작 시 필요한 아티팩트를 생성하는 도구이며, Orderer 를 위한 bootstrap block, Fabric 채널 설정 파일, Anchor peer 설정 파일로 구성돼 있다.

2.3.3. 인증서 및 아티팩트 생성

다음 명령을 통해 인증서를 생성한다. 아래 그림은 정상적으로 생성될 때 나타나는 메시지다.

```
cd $HOME/fabric-samples/first-network
../bin/cryptogen generate --config=./crypto-config.yaml
```



```
mjkong@bcdev:~/fabric-samples/first-network$ ../bin/cryptogen generate --config=./crypto-config.yaml
org1.example.com
org2.example.com
```

그림 2-2 Cryptogen 툴에 의해 인증서 생성 로그 화면

다음으로 configtxgen 툴을 이용해 orderer genesis block 을 생성한다.

```
FABRIC_CFG_PATH=$PWD
../bin/configtxgen -profile TwoOrgsOrdererGenesis -outputBlock ./channel-artifacts/genesis.block
```

다음으로 채널 트랜잭션 아티팩트를 생성한다.

```
## 테스트를 위한 <channel-ID>는 mychannel 입력하고, 계속 사용되므로 기억해 둔다.
export CHANNEL_NAME=mychannel
../bin/configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel-artifacts/channel.tx -channelID
$CHANNEL_NAME
```

다음으로는 Org1 과 Org2 에 대한 Anchor peer 아티팩트를 생성한다.

```
## <channel-ID>는 적절히 입력하고, 계속 사용되므로 기억해 둔다.
../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org1MSPanchors.tx -
channelID $CHANNEL_NAME -asOrg Org1MSP

../bin/configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./channel-artifacts/Org2MSPanchors.tx -
channelID $CHANNEL_NAME -asOrg Org2MSP
```

3. Fabric 네트워크 시작

앞에서 네트워크 시작을 위한 아티팩트의 생성을 완료했으며, 이제 Fabric 네트워크를 시작할 차례다. 시작을 위한 컴포넌트는 Orderer 1 개, Endorser 4 개(두 개의 Org 에서 각각 2 개의 피어), CLI 다.

여기서는 Docker 컨테이너를 관리하기 쉽도록 Docker compose 를 이용할 것이며, 관련 파일은 docker-compose-cli.yaml 이다.

우선 docker-compose-cli.yaml 파일을 텍스트 편집기로 열어서 70 번째 줄의 command 부분을 주석으로 처리한다.

```

53 cli:
54   container_name: cli
55   image: hyperledger/fabric-tools
56   tty: true
57   environment:
58     - GOPATH=/opt/gopath
59     - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
60     - CORE_LOGGING_LEVEL=DEBUG
61     - CORE_PEER_ID=cli
62     - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
63     - CORE_PEER_LOCALMSPID=Org1MSP
64     - CORE_PEER_TLS_ENABLED=true
65     - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/tls/tls.crt
66     - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/tls/tls.key
67     - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/tls/tls.ca.crt
68     - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
69   working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
70   #command: /bin/bash -c './scripts/script.sh ${CHANNEL_NAME} ${DELAY} ${LANG}; sleep 1000'
71   volumes:
72     - /var/run:/host/var/run/
73     - ../../chaincode:/opt/gopath/src/github.com/chaincode
74     - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto
75     - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/
76     - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/peer/channel-artifacts

```

그림 3-1 docker-compose-cli.yaml

그런 다음 아래 명령을 통해 네트워크를 시작한다.

```

## <channel-id>에는 앞서 입력한 채널 ID 값을 입력하고 TIMEOUT 에 적절한 시간(초)을 입력
CHANNEL_NAME=<channel-id> TIMEOUT=<pick_a_value> docker-compose -f docker-compose-cli.yaml up -d

```

3.1. 채널 생성

Docker 컨테이너가 정상적으로 시작됐으며, 다음과 같이 6 개의 컨테이너가 생성된다.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
bb804ffba251	hyperledger/fabric-tools	"/bin/bash"	12 seconds ago	Up 11 seconds		cli
ce3b6bc1dc57	hyperledger/fabric-peer	"peer node start"	16 seconds ago	Up 13 seconds	0.0.0.0:7051->7051/tcp, 0.0.0.0:7053->7053/tcp	peer0.org1.example.com
0cad469b41d0	hyperledger/fabric-peer	"peer node start"	16 seconds ago	Up 11 seconds	0.0.0.0:8051->7051/tcp, 0.0.0.0:8053->7053/tcp	peer1.org1.example.com
e12f1d3a7397	hyperledger/fabric-peer	"peer node start"	16 seconds ago	Up 11 seconds	0.0.0.0:10051->7051/tcp, 0.0.0.0:10053->7053/tcp	peer2.org1.example.com
087baf8246d1	hyperledger/fabric-orderer	"orderer"	16 seconds ago	Up 12 seconds	0.0.0.0:7050->7050/tcp	orderer.example.com
964cfb3007da	hyperledger/fabric-peer	"peer node start"	16 seconds ago	Up 14 seconds	0.0.0.0:7051->7051/tcp, 0.0.0.0:7053->7053/tcp	peer0.org1.example.com

그림 3-2 Fabric 네트워크 컨테이너 리스트

그리고 CLI 를 통해 채널 생성 및 체인코드와 관련된 명령어를 실행하기 위해서는 각 피어에 대한 환경변수 값을 알고 있어야 하며, 다음의 환경변수들을 앞으로 나올 명령에서 활용한다.

```

## peer0.org1
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/

```

```
org1.example.com/peers/peer0.org1.example.com/tls/ca.crt
```

peer1.org1

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp  
CORE_PEER_ADDRESS=peer1.org1.example.com:7051  
CORE_PEER_LOCALMSPID="Org1MSP"  
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/ca.crt
```

peer0.org2

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp  
CORE_PEER_ADDRESS=peer0.org2.example.com:7051  
CORE_PEER_LOCALMSPID="Org2MSP"  
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

peer1.org2

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp  
CORE_PEER_ADDRESS=peer1.org2.example.com:7051  
CORE_PEER_LOCALMSPID="Org2MSP"  
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
```

채널 생성을 위해 다음 명령을 통해 CLI 컨테이너로 접속한다.

```
docker exec -it cli bash
```

다음과 같은 형태로 프롬프트를 확인할 수 있다.

```
root@5670ce2f4dd2:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

CLI 컨테이너로 접속했으면 다음 명령을 통해 채널을 생성한다.

아래 명령이 정확하게 실행됐으면 아래 그림과 같이 <channel name>.block 파일이 만들어지는 것을 확인할 수 있다(테스트를 위해 앞서 입력했던 channel name 인 mychannel 로 환경 변수 설정).

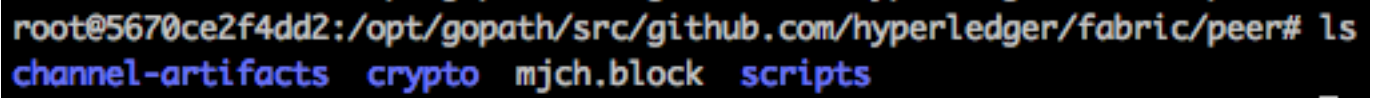
CHANNEL_NAME 을 환경변수로 설정합니다.

```
export CHANNEL_NAME=채널 명
```

CHANNEL_NAME 은 앞서 입력했던 chnnel id 와 동일하게 입력합니다.

```
peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/channel.tx --tls
```

```
$CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```



```
root@5670ce2f4dd2:/opt/gopath/src/github.com/hyperledger/fabric/peer# ls
channel-artifacts  crypto  mjch.block  scripts
```

그림 3-3 생성된 genesis 블록파일

3.2. 채널에 피어 참여

각 피어들을 채널에 참여시키기 위해서는 관련 환경변수가 각 채널에 맞게 설정돼야 한다. 그러한 항목으로 CORE_PEER_MSPCONFIGPATH, CORE_PEER_ADDRESS, CORE_PEER_LOCALMSPID, CORE_PEER_TLS_ROOTCERT_FILE 이 있다. 앞서 3.1 에서 명시한 각 피어별 환경변수 값을 활용한다.

3.2.1. Peer0.org1

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ADDRESS=peer0.org1.example.com:7051
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/ca.crt

peer channel join -b $CHANNEL_NAME.block
```

3.2.2. Peer1.org1

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
CORE_PEER_ADDRESS=peer1.org1.example.com:7051
CORE_PEER_LOCALMSPID="Org1MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.com/tls/ca.crt

peer channel join -b $CHANNEL_NAME.block
```

3.2.3. Peer0.org2

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
CORE_PEER_ADDRESS=peer0.org2.example.com:7051
CORE_PEER_LOCALMSPID="Org2MSP"
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/
```

```
org2.example.com/peers/peer0.org2.example.com/tls/ca.crt
```

```
peer channel join -b $CHANNEL_NAME.block
```

3.2.4. Peer1.org2

```
CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp
```

```
CORE_PEER_ADDRESS=peer1.org2.example.com:7051
```

```
CORE_PEER_LOCALMSPID="Org2MSP"
```

```
CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/peers/peer1.org2.example.com/tls/ca.crt
```

```
peer channel join -b $CHANNEL_NAME.block
```

이제 모든 피어가 채널에 참여하게 됐으며, 앞으로 각 피어별로 명령을 실행하기 위해서는 앞서 채널에 참여할 때 명령한 것과 같이 적절하게 피어의 환경변수를 설정해야 한다.

3.3. Anchor 피어 설정

모든 피어가 채널에 참여했으면 각 조직(Org1, Org2)별로 Anchor 피어 설정을 하며, Peer0.org1, Peer0.org2 를 Anchor 피어로 업데이트한다.

Peer0.org1 환경변수 설정 후

```
peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/Org1MSPanchors.tx --tls $CORE_PEER_TLS_ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

Peer0.org2 환경변수 설정 후

```
peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f ./channel-artifacts/Org2MSPanchors.tx -tls $CORE_PEER_TLS_ENABLED --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem
```

3.4. 체인코드 디플로이 및 동작 확인

블록체인을 이용하는 애플리케이션에서는 SDK 를 사용하기는 하지만 결국은 체인코드의 인터페이스를 통해 데이터를 핸들링하게 된다. 이때 필요한 동작으로 최초 체인코드의 Deploy 를 비롯해 블록에 데이터를 쓰기 위한 Invoke, 조회를 위한 Query 가 있다.

3.4.1. 체인코드 디플로이

Peer0.org1 환경변수 설정 후

```
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/chaincode_example02/go
```

3.4.2. 체인코드 초기화

다음 명령을 통해 체인코드를 초기화한다. 체인코드를 초기화 할 때는 Endorsement 정책을 설정하게 되는데, 아래의 명령어 중 -P 옵션이 의미하는 바가 Endorsement 정책이다. 설정되어야 할 Endorsement 정책의 내용은 클라이언트에 의해 트랜잭션 제안을 할 때 검증받아야 할 피어들의 조건들이다. 피어는 트랜잭션을 검증하는 과정에서 가장 중요한 단계 중 하나가 체인코드를 가상으로 실행해 보는 것이다. 아래 명령어에서 명시된 조건은 트랜잭션이 Org1 또는 Org2 에 포함되는 피어들 중 하나에서만 서명을 받으면 정당한 트랜잭션으로 판단되는 내용이다. 또한 초기값으로 a 에 100, b 에 200 을 블록체인에 입력한다.

Peer0.org1 환경변수 설정 후

\$CHANNEL_NAME 은 채널 생성 시 입력한 이름으로 대체

```
peer chaincode instantiate -o orderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n mycc -v 1.0 -c
'{"Args":["init","a","100","b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"
```

3.4.3. Query

체인코드 초기화까지 완료되면 Query 동작을 통해 블록에 입력된 값을 조회할 수 있다. 아래 그림과 같이 쿼리의 결과로 100 이 조회된 것을 확인할 수 있다.

Peer0.org1 환경변수 설정 후

\$CHANNEL_NAME 은 채널을 생성할 때 입력한 이름으로 대체

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

```
root@5670ce2f4dd2:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mjch -n mycc -c '{"Args":["query","a"]}'
2017-05-22 14:10:58.856 UTC [msp] getMspConfig -> INFO 001 intermediate certs folder not found at [/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp/intermediatecerts]
2017-05-22 14:10:58.856 UTC [msp] getMspConfig -> INFO 002 crls folder not found at [/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp/crls: no such file or directory]
2017-05-22 14:10:58.857 UTC [msp] getMspConfig -> INFO 003 MSP configuration file not found at [/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp/config.yaml: no such file or directory]
2017-05-22 14:10:58.903 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing local MSP
2017-05-22 14:10:58.903 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining default signing identity
2017-05-22 14:10:58.903 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6208031A0C08F2E28BC90510..6D7963631A0A0A0571756571
2017-05-22 14:10:58.903 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: 2E7DB92F01485A407D19B958BC29ABCA0F3BF6C811D41EBC451770D166DFF
Query Result: 100
2017-05-22 14:10:58.910 UTC [main] main -> INFO 008 Exiting
```

그림 3-4 Query 트랜잭션 결과 화면

3.4.4. Invoke

Fabric v0.6 에서처럼 이 체인코드는 자금이체를 간략하게 표현한 로직이며, 아래 명령은 a 에는 10 을 빼고 b 에는 10 을 더하는 역할을 한다.

Peer0.org1 환경변수 설정 후

\$CHANNEL_NAME 은 채널을 생성할 때 입력한 이름으로 대체

```
peer chaincode invoke -o orderer.example.com:7050 --tls $CORE_PEER_TLS_ENABLED --cafile
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C $CHANNEL_NAME -n mycc -c
'{"Args":["invoke","a","b","10"]}'
```

3.4.5. Query

앞서 Invoke 에서 정상적으로 트랜잭션이 실행되면 a 에는 90 이 들어있을 것이다. 이는 Query 를 통해 확인할 수 있다. 그런데 지금까지는 Peer0.org1 에서만 트랜잭션을 발생시켰는데, 다른 피어에서도 정상적으로 동작하고, 원장도 복제됐는지도 함께 확인할 필요가 있다.

다른 피어에서 Query 를 실행하기 전에 앞서 실행한 단계를 자세히 살펴보면 Peer0.org1 에서만 체인코드를 디플로이했고 다른 피어에서는 디플로이하지 않았다. 그래서 디플로이 되지 않은 피어에서 Query 를 실행하기 위해서는 우선 체인코드 디플로이부터 수행해야 한다.

Peer1.org2 환경변수 설정 후

\$CHANNEL_NAME 은 채널을 생성할 때 입력한 이름으로 대체

```
peer chaincode install -n mycc -v 1.0 -p github.com/chaincode/chaincode_example02/go
```

```
peer chaincode query -C $CHANNEL_NAME -n mycc -c '{"Args":["query","a"]}'
```

```

root@5670ce2f4dd2:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mjch -n mycc -c '{"Args":["query","a"]}'
2017-05-22 14:21:43.694 UTC [msp] getMspConfig -> INFO 001 intermediate certs folder not found at [/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp/intermediatecerts]
2017-05-22 14:21:43.694 UTC [msp] getMspConfig -> INFO 002 crls folder not found at [/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp/crls: no such file or directory]
2017-05-22 14:21:43.695 UTC [msp] getMspConfig -> INFO 003 MSP configuration file not found at [/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/peerOrganizations/org2.example.com/users/Admin@org2.example.com/msp/config.yaml: no such file or directory]
2017-05-22 14:21:43.729 UTC [msp] GetLocalMSP -> DEBU 004 Returning existing local MSP
2017-05-22 14:21:43.729 UTC [msp] GetDefaultSigningIdentity -> DEBU 005 Obtaining default signing identity
2017-05-22 14:21:43.730 UTC [msp/identity] Sign -> DEBU 006 Sign: plaintext: 0AAC070A6208031A0C08F7E78BC90510...6D7963631A0A0A057175657
2017-05-22 14:21:43.730 UTC [msp/identity] Sign -> DEBU 007 Sign: digest: E98AA91C7782FDED073E5FCA826CCDCA12F8027C2193E1689630066CF0C9
Query Result: 90

```

그림 3-5 Invoke 트랜잭션에 의해 변경된 값의 조회 화면

명령을 실행한 후 결과값이 90 으로 나오는 것을 확인할 수 있다. Peer1.org2 는 채널에는 등록돼 있었으나 체인코드는 디플로이되지 않은 상태로 다른 피어에서 트랜잭션이 있어난 후에 체인코드가 디플로이되더라도 원장은 최신 내용으로 관리되고 있음을 확인할 수 있다.