SPIN Protocol

함수형 프로그래밍 기반 BApp 개발 사례 소개

문해민

1. BApp 개발의 현실

BApp 개발에 대한 현실

Application

쉽지 않음

* 비지니스 로직 작성
* 확장성을 겸비한 설계
* 기획에 따른 변경과 대응
* 로드맵과 일정

+

Blockchain

쉽지 않음

* 이기종의 시스템
* 새로운 러닝커브

=

BApp

....

* 개발 복잡도 증가
* 운영 난이도 증가
* 그들의 압박ㅁㅁ...

같은 기능 다른 로직

```
const tokenSend1 = async _ => {
  const signer = caver.klay.accounts.wallet.add('0xPrivateKey');
  const contractABI = require('KRC20.json');
  const contractAddr = '0xd0e5ca14f494b230440aa613e89a8c7691a79fc4'
  const contract = new caver.klay.Contract(contractABI, contractAddr);

  contract.methods.transfer(address, caver.utils.toPeb(1,"KLAY")).send(
    {
      from: signer.address,
      gas: 20000000,
      gasPrice: await caver.klay.getGasPrice()
    }
  ).then(a => console.log(a))
}
```

[Case #1]

```
const tokenSend2 = async _ => {
  const signer = caver.klay.accounts.wallet.add('0xPrivateKey');
  const contractAddr = '0xd0e5ca14f494b230440aa613e89a8c7691a79fc4'
  const to = signer.Address
  const amt = caver.utils.toPeb(1,"KLAY")
  const funcSig = caver.klay.abi.encodeFunctionSignature('transfer(address,uint256)');
  const encodeParams = caver.klay.abi.encodeParameters(['address', 'uint256'], [to, amt]);
  const data = funcSig + encodeParams.substring(2)

  caver.klay.sendTransaction(
    {
      from: signer.address,
      to: contractAddr,
      gas: 20000000,
      gasPrice: await caver.klay.getGasPrice(),
      type: 'SMART_CONTRACT_EXECUTION',
      data
    }
  ).then(a => log(a))
}
```

[Case #2]

코드의 일관성이 떨어지므로 "코드 품질 검증에 대한 객관성 저하"

특정 규약이 없기에, "특정 상황에서만 동작하는 명령형 코드"로 전락할 가능성이 높음

명령형 코드의 문제점

```
const userAccounts = [
  { name: "Mark", age: 10, privateKey: "0xPrivateKey1" },
  { name: "John", age: 22, privateKey: "0xPrivateKey2" },
  { name: "Tomy", age: 19, privateKey: "0xPrivateKey3" }
]


for(const account of userAccounts)
  if (account.age > 20)
    signers.push(caver.klay.accounts.wallet.add(account.privateKey))

for(const signer of signers) signedAddresses.push(signer.address)

console.log(signedAddresses)
```

< Signed users address 추출 >

> 코드의 일관성 및 가독성이 떨어짐

코드 리딩에 있어 불리함

> 코딩 컨벤션 및 규약 정의가 힘듬

로직의 다각화 가능성이 열려있음

> 코드 복잡도가 급격히 증가할 수 있음

일주일 뒤에 다시 보면 기억이 잘 안남

> 코드 재사용성이 떨어짐

코딩 생산성 저하와 직결될 수 있음

3. 함수형 프로그래밍이란?

함수형 프로그래밍이란?

01

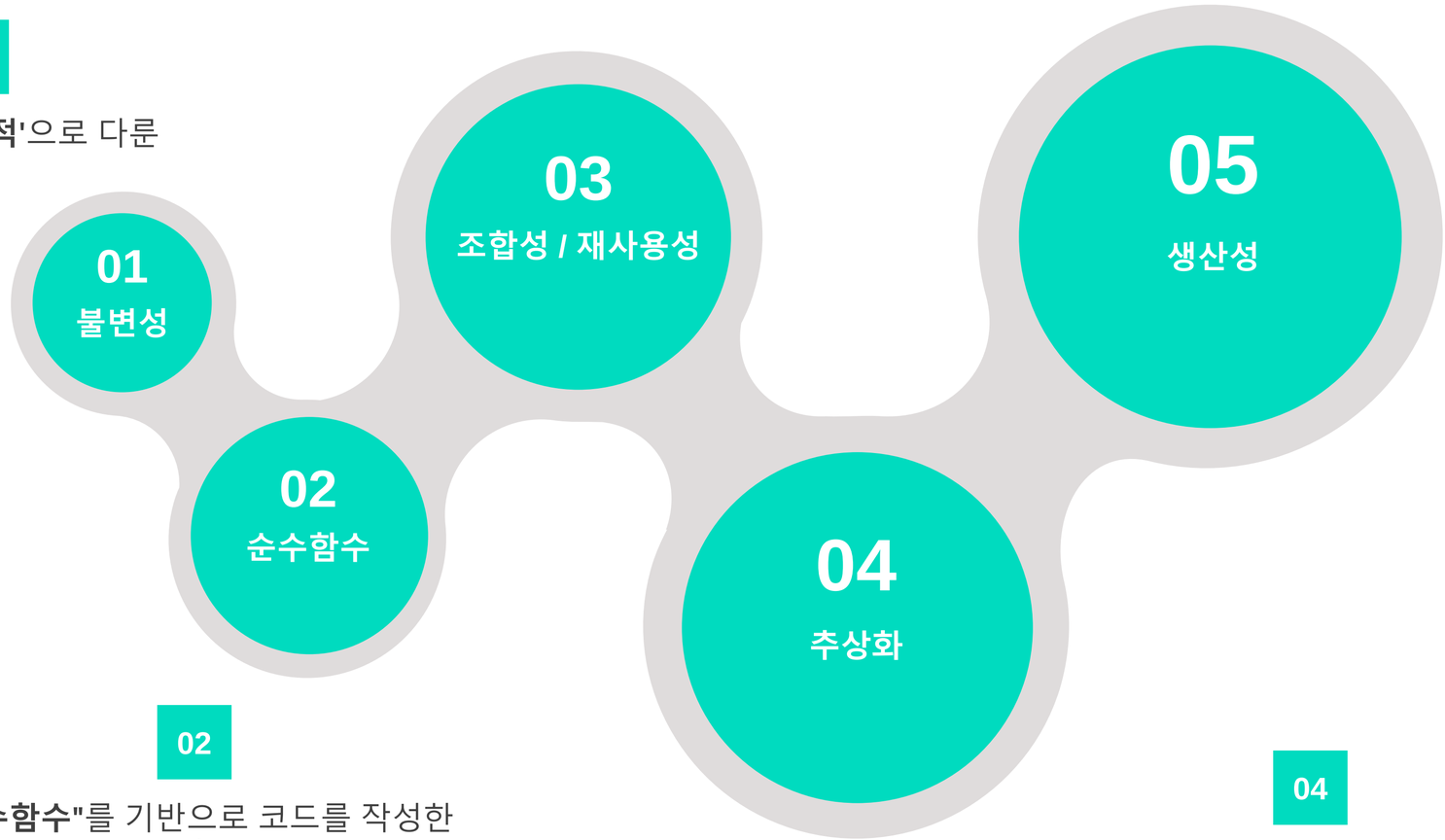
데이터를 '**불변적**'으로 다룬다.

03

순수함수들을 함수파라미터로 전달하여 새로운 **"고차함수"**를 작성한다.

05

"안전성, 조합성, 재사용성"을 강조하여 프로그래밍의 **"생산성"**을 높이고자 하는 패러다임을 갖고있다.



02

"순수함수"를 기반으로 코드를 작성한다.

04

추상화 레벨을 높여 높은 수준의 **"다형성"**을 제공한다.

4. Generic-caver

Generic-caver

함수형 프로그래밍의 규약과 패러다임을 접목시켜,

보다 Generic하게 caver.js를 활용해보자!

ffp-js

함수형 프로그래밍 기반

이터러블 프로그래밍 기반

지연성과 동시성을 지원

caver-js

자바스크립트의 표준 객체를 지원

기능별 모듈화 및 지역화가 잘 되어있음

메소드들이 순수함수 규약에 부합

안전하게 함수 합성 가능

규약에 의한 코드의 장점

```
const userAccounts = [
  { name: "Mark", age: 10, privateKey: "0xPrivateKey1" },
  { name: "John", age: 22, privateKey: "0xPrivateKey2" },
  { name: "Tomy", age: 19, privateKey: "0xPrivateKey3" }
]




go(
  userAccounts,
  filter(({ age }) => age > 20),
  ACCOUNTS.createSigners,
  ACCOUNTS.getSignedAddresses,
  log
)
```

> 안정성 및 조합성

파이프라인 함수 합성 기법을 통해
"안전하게 함수 조합이 가능"

> 데이터 불변성

원본 데이터를 유지하는 규약을 통해
"동시성을 보장해야하는 상황에 유리"

> 지연성

함수 선언형 코딩 방식을 통해
"메모리 릭을 방지하고 원하는 시점에 평가"

함수형 사고의 추상화 컨셉

```
const tokenSend1 = async _ => {
  const signer = caver.klay.accounts.wallet.add('0xPrivateKey');
  const contractABI = require('KRC20.json');
  const contractAddr = '0xd0e5ca14f494b230440aa613e89a8c7691a79fc4'
  const contract = new caver.klay.Contract(contractABI, contractAddr);

  contract.methods.transfer(address, caver.utils.toPeb(1,"KLAY")).send(
    {
      from: signer.address,
      gas: 20000000,
      gasPrice: await caver.klay.getGasPrice()
    }
  ).then(a => console.log(a))
}
```

"Method name을 통한 컨트랙트 호출"

[contract.methods.methodName(...inputs)]

```
const tokenSend2 = async _ => {
  const signer = caver.klay.accounts.wallet.add('0xPrivateKey');
  const contractAddr = '0xd0e5ca14f494b230440aa613e89a8c7691a79fc4'
  const to = signer.Address
  const amt = caver.utils.toPeb(1,"KLAY")
  const funcSig = caver.klay.abi.encodeFunctionSignature('transfer(address,uint256)');
  const encodeParams = caver.klay.abi.encodeParameters(['address', 'uint256'], [to, amt]);
  const data = funcSig + encodeParams.substring(2)

  caver.klay.sendTransaction(
    {
      from: signer.address,
      to: contractAddr,
      gas: 20000000,
      gasPrice: await caver.klay.getGasPrice(),
      type: 'SMART_CONTRACT_EXECUTION',
      data
    }
  ).then(a => log(a))
}
```

"Function signature를 통한 컨트랙트 호출"

[contract.methods[fnSignature](...inputs)]

두 방식의 차이점은?

함수 추상화의 장점

```
CONTRACT.write_feeDelegate = (signer, feePayer, contract, fnSignature, params) => go(
  inputGenerator(contract, fnSignature, params),
  inputs => fnCallGenerator(contract, fnSignature, inputs),
  encodedData => go(
    {
      from: signer.address,
      to : contract._address,
      data: encodedData,
      gasPrice: '25000000000',
      gas: 20000000,
      value: 0,
      nonce: KLAY.getNonce(signer.address),
      type: UTILS.txType('0x31')
    },
    unsignedData => ACCOUNTS.signTx(unsignedData, signer.privateKey),
    signedData => KLAY.sendRawTxFeeDelegated(signedData.rawTransaction, feePayer)
  )
);
```

```
go(
  CONTRACT.write_feeDelegate(
    adminSigner,
    feePayer,
    METADATA.Campaign,
    'createCampaign(uint256,uint256,uint256,uint256,uint256,uint256)',
    data
  ),
  tx => ({ status: true, receipt: tx }),
  success
);
```

< Generic-caver를 활용한 Fee delegation 예제 >

> 코드의 일관성 및 간결성 확보

코드 리딩에 있어서 유리함

> 재사용성이 높음

효율 및 생산성과 비례

> 응용력 향상

새로운 문제 해결에 용이

> 이기종 시스템간의 결합성이 높음

외부 유틸리티 툴의 도움을 받기에 용이

추상화 레벨을 높인 함수의 재사용성

```
/**
 * @param { QueryString } campaignId
 * @contract_call : Campaign
 */
exports.klaytn_getCampaign = async (event, context) => {
  if (event.source === 'bApp-warmer') return ;
  try {
    const data = convertEvent2inputData(event);

    return (!data)
    ? go({ status: false, message: 'Error params' }, failure)
    : go(
        data,
        params => CONTRACT.read(METADATA.Campaign, 'getCampaign(uint256)', params),
        pick([
          'productId',
          'revenueRatio',
          'totalSupply',
          'appliedInfluencers',
          'startAt',
          'endAt',
          'createdAt'
        ]),
        success
      );
  } catch (e) {
    return go({ status: false, message: e.message }, failure);
  }
}
```

```
exports.klaytn_getProductData = async (event, context) => {
  if (event.source === 'bApp-warmer') return ;
  try {
    const data = convertEvent2inputData(event);

    return (!data)
    ? go({ status: false, message: 'Error params' }, failure)
    : go(
        data,
        params => CONTRACT.read(METADATA.Product, 'getProductData(string,uint256)', params),
        pick([
          'viewCount',
          'purchaseCount'
        ]),
        success
      );
  } catch (e) {
    return go({ status: false, message: e.message }, failure);
  }
}
```

```
exports.klaytn_getRevenueLedgerList = async (event, context) => {
  if (event.source === 'bApp-warmer') return ;
  try {
    return go(
      CONTRACT.read(METADATA.RevenueLedger, 'getRevenueLedgerList()', {}),
      pick(['revenueLedgerList']),
      success
    );
  } catch (e) {
    return go({ status: false, message: e.message }, failure);
  }
}
```

5. 결론

FP는 규약과 패러다임을 익히는 것

01 패러다임의 성숙도 향상

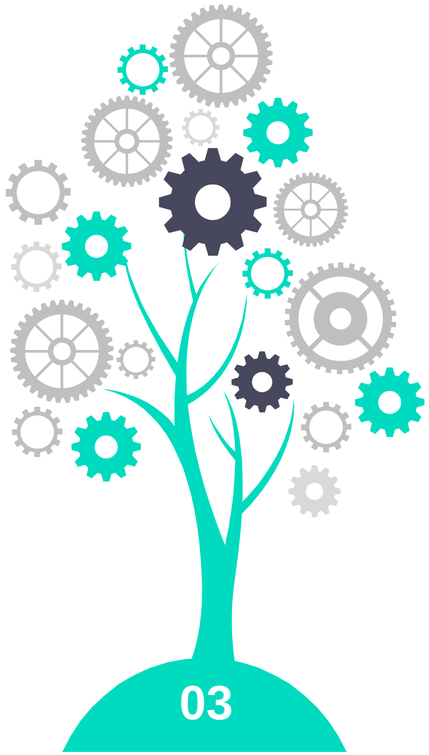
규약 중심의 작업은 구성원간 같은 관점을 갖게하여 소통력을 높이므로 **"협업에 용이"**

02 코드 품질에 대한 객관성

규약에 의한 코드는 **"코드의 일관성"**이 유지되므로, **"코드 품질의 객관성"**을 확보

03 생산성 향상

규약이 지켜진 코드는 **"안전성, 조합성, 재사용성"**이 높으며, 이는 **"프로그래밍의 효율과 생산성에 직결"**



감사합니다

harace@spinprotocol.io