[FAB-10741]
Transaction methods in chaincode

Abstract

This proposal adds the ability to define individual transaction processing functions as an alternative to a single ‘chaincode’ class with an Invoke() method and a dispatch table. The transaction functions will be discoverable by client applications, allowing for type-safe APIs to be generated.

An example of what a Contract would look like is in https://github.com/mbwhite/chaincode-commericalpaper

This is working with the code in cited repos, not a mock-up

Motivation

In the Fabric chaincode today, a class/interface is specified that has to implement two methods (Init & Invoke). The Invoke method handles all transaction and query requests for that chaincode instance and is passed a ‘stub’ argument which has a method that returns the transaction name and its arguments. Based on this information, the Invoke method can either handle the transaction logic itself or dispatch it to another method in the chaincode class. The logic is very dynamic and does not lend itself well to the types of static analysis that modern development tools use to assist the programmer.

The overriding aim here is to increase the efficiency for developers producing a production solutions; note that the word function is being used in a general sense.

Example node.js chaincode:

```
 // The Invoke method is called as a result of an application request to run the
Smart Contract  xxxxxx
  // The calling application program has also specified the particular smart
contract
  // function to be called, with arguments
```

```
async Invoke(stub) {
  let ret = stub.getFunctionAndParameters();
  console.info(ret);

  let method = this[ret.fcn];
  if (!method) {
    console.error('no function of name:' + ret.fcn + ' found');
    throw new Error('Received unknown function ' + ret.fcn + ' invocation');
     }
  try {
    let payload = await method(stub, ret.params);
    return shim.success(payload);
     } catch (err) {
    console.log(err);
    return shim.error(err);
     }
}
```

This example makes no reference to the specific Smart Contract - it's copied verbatim from the code - the only reference was in the comment. Hence this will need to be copied and reused by all coders. With JavaScript's approach the implementation is more succulent than in other languages, but still additional 'boilerplate' code that is required. .

Other blockchain platforms have transaction functions as a first class entity, for example Solidity. One of the benefits of explicit declaration of first class functions is the ability for tooling to more easily locate them, and provide support.

High-level design

Overview

An instance of ‘Chaincode’ has a 1:1 relationship with the instance of the running Docker container that is hosting the chaincode.

A ‘Contract’ is an abstract class/interface that the developer can extend/implement with a number of methods/functions. Within a Chaincode instance, there will be instances of a contracts. The creation/destruction of the Contract instance will follow that of the Chaincode.

Multiple instances of Contracts may exist, but must have different implementations.

Each instance of a Contract must be defined by a ‘namespace’. A single Chaincode instance scopes the namespace.

Inversion of Control
Today the chaincode has to insert itself into the shim, and subsequently start it.

```
shim.start(new Chaincode());
```

By adjusting the shim to load based on specific configuration, and conventions will simplify the approach. behaviour if required.

Backwards Compatibility
This is an alternative to the existing chaincode format that doesn’t affect the behavior of existing code; the initial invocation of the chaincode will be via the `npm start` command issued by the peer. Therefore any chaincode that is written today will start as per normal.

The library fabric-shim library will be useable as-is. A goal would be to able to have an API module that developers could you for design and testing, and then bring in the full implementation only when need with a real Fabric instance.

In documentation and code, the chaincode interface (as today) will be referred to as the SPI, and the new interface as the API. Both equally valid but the distinction is to refer to their different levels of abstraction.

Rationale
Experience with Hyperledger Composer has demonstrated the value of this form of transaction processing functions because of the type safety associate with them.

- All chaincode today has to include the routing logic. Simple examples such as fabcar and marbles this is a few functions that can be contained in a single file. A simple switch or array based approach is sufficient.

- Tools like chaintool also seek to help this issue by building on top of the init/invoke mechanism.

Details

Node.js

A set of contracts is packaged as a **node.js module** using the node.js standard process of defining entrypoints into the module. Specific elements of the code need to be indentified as providing the endpoints. Not all the functions within a set of code would wish to be exposed as entrypoints

There are multiple ways this could be done; there are advantages/disadvantages of all approaches - with no clear technically superior one. Hence it is best to provide a clear set of choices for developers to chose the one that fits best within their own architecture.

Here is an example flow of developer writing a new smart contract.

Prepare a new npm module

```
$ npm init -y

Add in support libraries
$ npm install --save fabric-contract-api
$ npm install --save <my-own-choice-of-extra-business-logic>
```

How to setup the Fabric shim to load this contract
(a) setup by declarative configuration/convention

A class **must** extend `SmartContract` as follows.

```
// SDK Library to asset with writing the logic
const { Contract } = require('fabric-contract-api');
Class MyContract extends Contract {

  constructor() {
      super(<optional namespace string>)
  }

  async myTransaction1(api, arg_1, arg_2, arg_3) {
  }

  async myTransaction2(api, arg_1, arg_2) {
```

```
  }

  async myQuery1(api, arg1) {
      return results;
  }
}
module.exports = MyContract;
```

The arguments are expanded from the request, rather than be passed as a single array

The standard node mechanism of defining the exports from a module can then be used
NOTE contracts in the exports is important; for future should something else need to be exported, ACLs, query definitions etc.

```
// index.js
const Contract= require('./mysmartcontract.js)

// export the smart contracts
module.exports.contracts = [Contract];
```

(a.i) *<note - deleting the function export declaration as unnecessary complications>*

(a.ii) By explicitly defining the class
Specify a name of the class to load either in the package.json, by exporting them from the module as a whole.

```
{
  "name":....
  "contracts": {
      "classes" : "mysmartcontract.js"
  }
}
```

(b) *By executing code and registering functions programmatically*
First you could call the same code that the bootstrap command does

```
require('fabric-shim').spi.startChaincode();
```

Or specifically give the classes

```
const UpdateValues = require('./updatevalues')
const RemoveValues = require('./removevalues')
require('fabric-shim').spi.register( [UpdateValues,RemoveValues] );
```

Starting the chaincode container

The code in the peer will issue `npm start` as per today. *This means that anything existing chaincode works as is today. Migration is therefore not a problem.*

The chaincode module can then either put `startChaincode` as the implementation of the the start script - or call one of the programmatic methods above. `startChaincode` is a binary that is provided to 'bootstrap' the contracts.

On instantiation, a docker image is built for this chaincode, the package.json and code copied in. and `npm install` run.

It is important to make sure that you have a `package-lock.json` to ensure the correct packages are imported.

Assume that `startChaincode` is defined as the script to run for npm start. The constructors of the exported Contracts will be run at this point; these constructors are for setting the namespace and optionally setup of the 'error/monitoring functions', (see below). This instance of the contract will existing whilst this chaincode docker image is up. There could be multiple instances of contracts

When chaincode is instantiated or updated, the init() function of the chaincode is called. As with the invoke() call from the client, a fn name and parameters can be passed. Remember therefore to have specific functions to call on init() and update() in order to do any data initialization or migration that might be needed. These two functions have been abstracted away to focus on specific function implementations.

Additional Functional Richness

The addition of a Contract class permits the ability to easily additional richness. Specifically the following will be present

- A function can be defined by the user to be invoked if an unknown function request comes in. The default would be to throw an exception.
- A single function can also be defined that could be invoked prior to the requested invoke function. This would enable logging for example (default would be a neutral - do nothing function). It also permits additional information to be passed onto the transaction function.
- A single function that can also be defined that is call post transactions being executed… this is passed the result of the function. This is ONLY called if the function has not failed with an error.

Chaincode API

Within a function defined in the contract, the code can process the business logic that is required.

The inputs to any given function f, are a transaction context and the arguments that have been passed from the invoking client SDK

The arguments are passed “as-is” to the function - no modification or validation is performened.

The transaction context provides access to both information about the current transaction, and the API that can be used by the business logic.

The ‘context’ (abbreviated to ctx in code) is an object that has two properties, ‘stub’ which is an instance of the existing stub API, and clientIdentity which is an instance of the ClientIdentity object created.

In addition there is a ‘getCallData()/setCallData()’ pair of functions - this can be used to set/get an object in the before/after functions and provide a route to pass data between them.

This context object does allow future freedom of movement if additional information or APIs become available.

NB. there is some debate as to the merits of properties on objects being directly accessed vs accessor methods. Different languages as well favour different idioms. Hence for some languages the process to get the API might vary.

Go

Defining a contract for chaincode

The struct must embed 'contractapi.contract' within it. This defines it as a valid contract for use in chaincode and provides a number of generic functions. The user then can add functions to this contract in the traditional Go manner.

Defining a function for a contract for chaincode

The functions that the user wishes to make callable through the chaincode (via an init, invoke or query) can have any name however they must be made public for use outside the package. Functions that are not public will not be available to callers of the chaincode, this allows the user to define private functionality for a contract to use internally. The functions that are private can be of any format but the functions for use in chaincode may only take in:

- **TransactionContext** if used it must be the first parameter the function takes in.
- **string** - functions can take 0 or more string parameters. When a call passes in the argument list the values are assigned to the string paremters in order such that the first value in the argument list is assigned to the first string parameter in the function definition. If a call does not provide enough values in the arg list then the blank string is provided to string parameters after the first n. If a user provides an arg list longer than the number of parameters the first n are for named string parameters in the function and the rest are ignored unless a []string is used in the function definition.
- **[]string** – if used it must be the last parameter the function takes in. The string slice will contain any values passed in via the call to chaincode that are not taken by named string parameters e.g. a function takes in 3 named string parameters but 5 values are passed in the arg list the first 3 values will be used for the named parameters and the string slice will contain the remaining 2. If no string parameters are defined in the function the entire arg list will be present in the slice.

Functions for used in chaincode may only return the following:

- **<nothing>**
- **string**
- **error**
- **string, error**

If the function is not defined to return an error or returns a nil value for the returned error shim.Success will be returned for the chaincode call with the returned string value (if any), if no

string value is defined to be returned the blank string is used for the success. If an error value is returned and not nil then shim.Error will be returned for the call.

Starting a chaincode container using contracts

To start the chaincode the user calls `contractapi.CreateNewChaincode` in their main function replacing the call to shim.Start in traditional chaincode. They pass to this call the set of contracts they wish to use in their chaincode, the contract converts these to a single chaincode and calls shim.Start on this generated chaincode. This chaincode stores a copy of the contracts rather than a reference meaning if the user updates the original passed in contract this will not be reflected in the chaincode. If multiple contracts are being passed each must have a unique namespace set before the call is made. This can be set using the SetNamespace() function provided by the embedded contract struct. If no namespace is given a default namespace of 'contract' is used. Passing 'bad' contracts to this function will cause a panic.

Example startup:

```
func main() {
    sa := SimpleAssetContract{}
    sa.SetNamespace("simpleasset")

    ca := ComplexAssetContract{}
    ca.SetNamespace("complexasset")

    if err := contractapi.CreateNewChaincode(&sa, &ca); err != nil {
        fmt.Printf("Error creating/starting chaincode: %s", err)
    }
}
```

Calls to a contract function via chaincode

Calls to contract functions inside chaincode are formatted in the same way as a regular call to chaincode however the args list passed must contain the contract namespace followed by the contract function name as the first parameter. Further parameters are used as values to be passed to the function once called.

Example invoke:

```
peer chaincode invoke -n mycc -c '{"Args":["simpleasset_Manage", "abc123", "20"]}' -C myc
```

The above calls the function 'Manage' in the contract with the namespace 'simpleasset'. The values 'abc123' and '20' are passed to the function if the function is defined to require them.

Additional Functions

- A function can be defined to be called when a call to the chaincode contains the name of a function that is not defined for the contract with that namespace. This function does not have to be part of the contract or publicly available outside the package but must follow the format for functions that can be called as part of chaincode. If this function returns an error shim.Error is returned for the call with that error. If it returns a value shim.Success is returned with that value. If no unknown function is defined then shim.Error is returned with a message that the named function given in the call was not found.
- A function can be defined to be called before each call to the chaincode for a given contract's namespace. This function does not have to be part of the contract or publicly available outside the package but must follow the format for functions that can be called as part of chaincode. If the function returns an error the named function passed is not called and shim.Error is called with this functions error. If the function does not return an error or it returned an error value of nil its returned value is ignored.
- A function can be defined to be called after each call to the chaincode for a given contract's namespace. This function does not have to be part of the contract or publicly available outside the package but must follow the format for functions that can be called as part of chaincode. If a previous function be it the before function, named function in the call or unknown function errors this after function is not executed. If this after function returns an error then that error is returned via shim.Error, if the function is not defined to return an error or it returns an error value of nil the output of the function is ignored and the named or unknown function's output is returned.

*Note: All additional functions will use the **same** arguments as passed into the Init/Invoke.*

Contract API

Contract

Exports functions such that it is compliant with ContractInterface. Required to be embedded in contracts to be converted to chaincode. Has no public properties but private properties store the unknown before and after functions, the contract's namespace, whether it is inside chaincode and its metadata.

SetUnknownFn()

Takes a function passed in and stores it as the unknownFn as a *contractFunction* which stores a reference to the function and details about what it takes in and returns. It provides a call method so that the original function can be called. Used for setting the function to be called when a call is made to the contract with the name of a function that doesn't exist (publicly). Function passed must match patterns defined for a valid contract

function but do not have to be public or hang off the contract. Will panic if function passed it does not match valid contract function pattern.

GetUnknownFn()

Returns the current set unknown function's call method. The call method is not the actual function but another which calls the original function. As such calling the function using this value means passing in the transaction context and any number of string arguments rather than the original functions defined structure. Returns error when function not set. Return format; (func(TransactionContext, ...string) (string, error), error)

SetBeforeFn()

Takes a function passed in and stores it as the beforeFn as a *contractFunction* which stores a reference to the function and details about what it takes in and returns. It provides a call method so that the original function can be called. Used for setting the function to be called before the named call when a call is made to the contract through chaincode. Function passed must match patterns defined for a valid contract function but do not have to be public or hang off the contract. Will panic if function passed it does not match valid contract function pattern.

GetBeforeFn()

Returns the current set before function's call method. The call method is not the actual function but another which calls the original function. As such calling the function using this value means passing in the transaction context and any number of string arguments rather than the original functions defined structure. Returns error when function not set. Return format: (func(TransactionContext, ...string) (string, error), error)

SetAfterFn()

Takes a function passed in and stores it as the afterFn as a *contractFunction* which stores a reference to the function and details about what it takes in and returns. It provides a call method so that the original function can be called. Used for setting the function to be called after the named call when a call is made to the contract through chaincode. Function passed must match patterns defined for a valid contract function but do not have to be public or hang off the contract. Will panic if function passed it does not match valid contract function pattern.

GetAfterFn()

Returns the current set after function's call method. The call method is not the actual function but another which calls the original function. As such calling the function using this value means passing in the transaction context and any number of string arguments rather than the original functions defined structure.Returns error when function not set. Return format: (func(TransactionContext, ...string) (string, error), error)

SetNamespace()

Sets the namespace of the contract. A contract cannot have its namespace set if it is inside chaincode so the namespace must be set already on the contract passed in to the create chaincode function if there is an attempt to do so it will panic.

GetNamespace()

Returns the current set namespace. If no namespace has been set it returns the default namespace value. Return format: string

GetContractMetadata()

Returns the current set metadata string. The metadata can only be set by the contractapi itself and is set when a contract is used for chaincode as a JSON string of *MetadataContractChaincode*. Each contract stores only metatadata about itself. The system contract created automatically when new chaincode is created using contracts stores metadata about all contracts making up the chaincode. Return format: string

Example contract metadata (formatted):

```
{
    "namespaces": {
       "simpleasset": {
             "contract": {
                   "namespace": "simpleasset"
             },
             "functionNames": ["Get", "GetContractMetadata", "Manage"]
       }
    }
}
```

Example system contract metadata (formatted):

```
{
    "namespaces": {
       "complexasset": {
             "contract": {
                   "namespace": "complexasset"
             },
             "functionNames": ["Create", "Get", "GetContractMetadata",
"GetOwner", "GetValue", "SetOwner", "UpdateValue"]
       },
       "org.hyperledger.fabric": {
             "contract": {
                   "namespace": "org.hyperledger.fabric"
```

```
            },
            "functionNames": ["GetContractMetadata"]
        },
        "simpleasset": {
            "contract": {
                "namespace": "simpleasset"
            },
            "functionNames": ["Get", "GetContractMetadata", "Manage"]
        }
    }
}
```

MetadataContractChaincode

Has property *Namespaces* storing map of MetadataNamespace. JSON enabled so that a string JSON of a metadata for a chaincode contract can be parsed to a useful entity in Go. JSON created for metadata in a contract is generated using this struct.

MetadataNamespace

Has properties *Contract* and *FunctionNames*. *Contract* stores a MetadataContract. *FunctionNames* stores string array. JSON enabled so that a string JSON of a metadata for a namespace can be parsed to a useful entity in Go.

MetadataNamespace

Has property *Namespace* storing a string. JSON enabled so that a string JSON of a metadata for a specific contract can be parsed to a useful entity in Go.

TransactionContext

Created when Init/Invoke called on chaincode where it stores the stub passed into Init/Invoke, the clientIdentity from that stub and empty callData. All properties private and only retrievable via accessor functions. Only callData has accessor function for updating its value. Passed to contract functions when function defines it should be.

GetStub()

Returns stub property. Return format: shim.ChaincodeStubInterface

GetClientIdentity()

Returns clientIdentity property. Return format: cid.ClientIdentity

SetCallData()

Allows the user to set the call data property to any value and type. This value can then be shared between the function and other functions during the transaction. E.g. the

before function can set it and then named function use the value and update it for the after function to finally use. Helpful in getting around being unable to read own writes.

GetCallData()

Returns callData property. Value will be returned as interface type so user will need to cast it to something useful once they have retrieved it.

CreateNewChaincode(contract1, contract2, ...contractN)

Takes in series of contracts that match the contractInterface and converts them to a chaincode it starts up. Will panic if there is an error during conversion e.g. clashing namespaces.

Test Cases

Unit tests and end to end tests will be created to cover all code paths
FV testing of end to end codepaths for both existing and new styles of chaincode

Copyright