TENEMOS
MUCHO
QUE HACER
JUNTOS

tecnalia Inspiring Business

3 - Hyperledger Fabric GO chaincode development

Introducción

Desarrollo de smart contracts en GO

Introducción

Qué es un **smart contract / chaincode**?

```
xxd ./contract.out | head

00000010: 7a69 1a05 312e 302e 301a 001a fb8b 441f  zi..1.0.0.....D.
00000020: 8b08 0000 0000 0000 ffec bd7b 7b1a 39b2  ...........{{.9.
00000030: 307e fe85 4f51 d3ef 9309 2404 9a8b b1c3  0~..OQ....$.....
00000040: acf7 fd11 9b64 38c7 c679 8d33 b37b 3279  .....d8..y.3.{2y
00000050: bca2 5b80 264d 37d3 ddf8 12af bffb ef51  ..[.&M7........Q
00000060: 49ea 56df 005f e2b9 6c78 661c e896 4aa5  I.V.._..lxf...J.
00000070: aa52 a954 2a95 02df 6acc c917 d6b0 3c37  .R.T*...j.....<7
00000080: 0889 1b06 0ddf 7368 509f 79ff f558 1fb3  ......shP.y..X..
00000090: 699a dd4e e7bf 4cd3 3477 bbc9 7f4d d36c  i..N..L.4w...M.l
```

Introducción

Qué es un **chaincode**?

Chaincode is a program, written in GO, node.js, and eventually in other programming languages such as Java, that implements a prescribed **interface**. Chaincode runs in a **secured Docker** container isolated from the endorsing peer process. Chaincode initializes and manages ledger state through transactions submitted by applications.

1. **Qué es :** una aplicación software que se ejecuta en un servidor.
2. **Dónde se ejecuta:** en los nodos endorser de **HL Fabric.**
3. **Cómo:** mediante contenedores docker aislados con un máx de 2Gb de RAM (por defecto).

Entorno de desarrollo

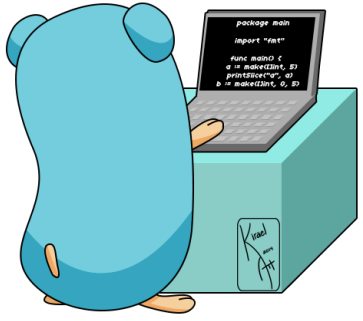
Para desarrollar un smart contract en GO se necesita de al menos:

- Un entorno de desarrollo y **PRUEBAS**! (Sublime Text, VCode, Goland, etc)
- Entorno de desarrollo de GO.

```
sudo apt-get install golang-go
```

- Conexión a Internet
- Documentacion del SDK de Fabric: https://godoc.org/github.com/hyperledger/fabric-sdk-go
- https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim

Mi entorno de desarrollo favorito



Go

Lenguaje de programación

GoLang: Introducción

- Go es un lenguaje moderno, por lo que bebe de lo mejor de muchos lenguajes. Combina una sintaxis parecida a **`C`** con las características y facilidad de lenguajes dinámicos como **`Python`**. Lenguajes como **`C++`**, **`Java`** o **`C#`** son más pesado o voluminosos.

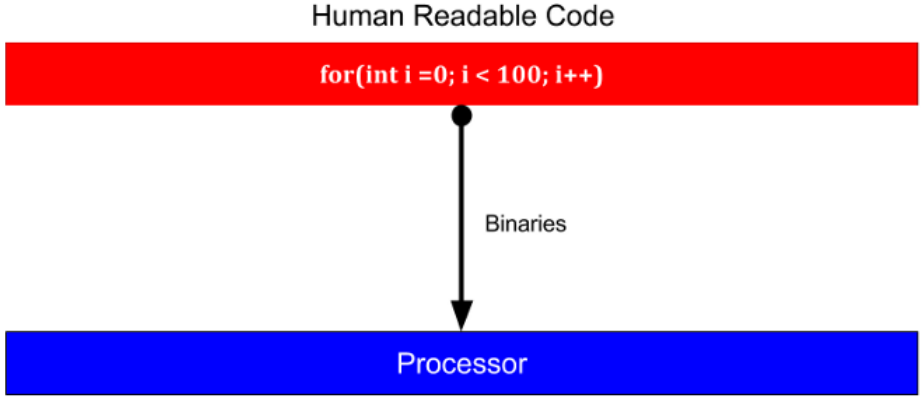


Imagen: https://medium.com/exploring-code/why-should-you-learn-go-f607681fad65

GoLang: Introducción - Ejemplos

- En cambio, GO acierta con una sintaxis clara y concisa.

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

Hello world en Go

GoLang: Introducción - Función simple

- Ejemplo de una función que ejecuta una suma de dos operandos.

```
package main

import "fmt"

func add(x int, y int) int {
    return x + y
}

func main() {
    fmt.Println(add(42, 13))
}
```

Función suma

GoLang: Introducción - Funciones

- Ejemplo de una función que ejecuta una suma y multiplica dos operandos, y devuelve ambos resultados.

```
package main

import "fmt"

func addMultiply(x int, y int) (int, int) {
    return x + y, x * y
}

func main() {
    fmt.Println(addMultiply(42, 13))
}
```

Función suma y multiplicación

GoLang: Introducción - Funciones

- Ejemplo de una función que ejecuta una suma y multiplica dos operandos (usando variables) y devuelve ambos resultados.

```
package main

func addMultiply(x int, y int) (int, int) {
    suma := x + y
    multiplicacion := x * y
    return suma, multiplicacion
}

func main() {
    sum, mult := addMultiply(42, 13)
}
```

Función suma y multiplicación con variables

GoLang: Introducción - Paquetes

1. Go, como cualquier otro lenguaje de programación se organiza en paquetes. Estos paquetes se pueden descargar con el gestor de paquetes que incorpora go, mediante el comando **`go get package_name`**
2. En Go, los paquetes se importan de 2 formas dependiendo del número de paquetes necesarios. A continuación se muestran las diferencias:

```
package main

import "fmt"
```

```
package main

import (
    "fmt"
    "logger"
)
```

Importando paquetes en Go

GoLang: Introducción - Control

Las estructuras de control disponibles son: if, if-else, switch y for. Por lo tanto no existen el formato **`while`** como ocurre en otros lenguajes de programación.

```
func main() {
    fmt.Print("Go runs on ")
    switch os := runtime.GOOS; os {
    case "darwin":
          fmt.Println("OS X.")
    case "linux":
          fmt.Println("Linux.")
    default:
          // freebsd, openbsd,
          // plan9, windows...
          fmt.Printf("%s.", os)
    }
}
```

Ejemplo de instrucción **SWITCH** en **GO**

GoLang: Introducción - Estructuras

En **GO**, lo que en **Java** se conoce como **Clase**, se denomina **struct** y se definen de la siguiente forma:

```
type person struct {
    name string
    age  int
}
```

Ejemplo de una struct en **GO**

GoLang: Introducción - Acceso

En GO, se definen dos tipos de acceso principalmente: **público o privado.** las funciones que son **privadas** deberán empezar siempre por letra minúscula, y las **públicas** por mayúscula.

Lo mismo ocurre con los atributos de una `struct`

GoLang: Introducción - Resumen

1. Go es un lenguaje compilado.
2. Incluye su propio garbage collector.
3. Posibilidad de usar `struct` como en `C/C++` y punteros.
4. Orientado a alto rendimiento y seguridad.
5. Incluye herramientas nativas de benchmarking, test y race detection.
6. Reflection.
7. Simplicidad de implementar hilos. Toma lo mejor de otros lenguajes.
8. Fácil de implementar la concurrencia y fácil de gestionarla.

Pero... no es oro todo lo que reluce y `GO`...

GoLang: Introducción - Resumen

Tiene las siguientes ‘desventajas’ frente a otros lenguajes:

- No soporta herencia
- No tiene constructores
- No tiene anotaciones
- No tiene Generics
- No tiene excepciones

Estructura

Arquitectura de un smart contract

Arquitectura de un smart contract

- Un smart contract se divide en 3 partes diferentes
  - **La invocación**: Un smart contract sólo se invoca una única vez por nodo y canal.

```
func Start(cc Chaincode) error
```

  - **La inicialización:** El smart contract ejecuta una función especial llamada `Init(stub)` que se llama la primera vez que se ejecuta.

```
Init(stub ChaincodeStubInterface) pb.Response
```

  - **La ejecución de transacciones:** Cada vez que se realiza una transacción, ésta se recibe a través del método `Invoke(stub)`

```
Invoke(stub ChaincodeStubInterface) pb.Response
```

Start(cc)

Chaincode **Start** call. All Chaincodes must inherit a Chaincode interface in order to properly interact with the blockchain and with the ledger. The fabric runs the transactions by calling these functions as specified.

Es la unica funcion que tiene que tener en cuenta el desarrollador. Si esta función no se llama de forma explícita, el smart contract no se ejecutará.

Init(stub)

In particular the **Init** method is called when a chaincode receives an **instantiate** or **upgrade** transaction so that the chaincode may perform any necessary initialization, including initialization of application state.

Invoke(stub)

The **Invoke** method is called in response to receiving an invoke transaction to process transaction proposals.

Fabric SDK

Puntos clave

tecnalia Inspiring Business

package shim

import "github.com/hyperledger/fabric/core/chaincode/shim"

Package **`shim`** provides APIs for the chaincode to access its state variables, transaction context and call other chaincodes.

GetArgs() [][]byte

`GetArgs` returns the arguments intended for the chaincode **`Init`** and **`Invoke`** as an array of byte arrays.

Devuelve el contenido de los parámetros de las peticiones que se realizan en formato de byte array. Dicho bytearray, tendrá que ser procesado manualmente más adelante.

GetStringArgs() []string

`GetStringArgs` returns the arguments intended for the chaincode Init and Invoke as a string array. Only use GetStringArgs if the client passes
arguments intended to be used as strings.

Devuelve el contenido de los parámetros de las peticiones en formato String.

Cuidado con este metodo! Si recibe parámetros de tipo, **`int`**, **`float`**, **`double`**...fallará

GetTxID() string

`GetTxID` returns the tx_id of the transaction proposal, which is unique per transaction and per client. See ChannelHeader in protos/common/common.proto for further details.

Devuelve el identificador **único** de la transacción que se está realizando en dicho momento. Ejemplo:

```
e4f7e564b7ab7e42753dfc291b03e19ffc63c4b63f4010a41ffe13730ef63a7a
```

GetChannelID() string

`GetChannelID` returns the channel the proposal is sent to for chaincode to process. This would be the channel_id of the transaction proposal (see ChannelHeader in protos/common/common.proto) except where the chaincode is calling another on a different channel

Devuelve el identificador **único** del canal en el que se está ejecutando el smart contract.

```
main
```

PutState(key, value) error

PutState puts the specified **`key`** and **`value`** into the transaction's writeset as a data-write proposal. PutState doesn't effect the ledger until the transaction is validated and successfully committed. Simple keys must not be an empty string and must not start with null character (**`0x00`**), in order to avoid range query collisions with composite keys, which internally get prefixed with **`0x00`** as composite key namespace.

tecnalia Inspiring Business

GetState(key) []byte, error

GetState returns the value of the specified **`key`** from the ledger. Note that GetState doesn't read data from the **`writeset`**, which has not been committed to the ledger. In other words, GetState doesn't consider data modified by PutState that has not been committed. If the key does not exist in the state database, (nil, nil) is returned.

DelState(key) error

`DelState` records the specified **`key`** to be deleted in the writeset of the transaction proposal. The **`key`** and its value will be deleted from the ledger when the transaction is validated and successfully committed.

Fabric SDK

Modelo de datos e interacción

Interacción con el ledger y state

Fabric admite las siguientes formas de escritura de información (por defecto):

- leveldb (key-value de toda la vida)
- couchdb (base de datos orientada a documentos, formato json, indices, ordering, paginado, etc)

- Cuando se usa leveldb, se realiza mediante:
  - `stub.PutState(string, []byte), stub.GetState(string), stub.DelState(string)`

- Cuando se usa couchdb, se realiza mediante:
  - `stub.PutState(string, []byte), stub.GetQueryResult(string)`

Modelo de datos y struct

```
// Simplest struct ever
type SimpleStruct struct {
}
```

Veamos un ejemplo...

```
type Persona struct {
    nombre string
    direccion string
    dni string
    edad int
}
```

Veis algo raro?

Modelo de datos

Si procedemos a serializar la información como JSON, obtendremos el siguiente resultado:

```
{}
```

Ninguno de los campos declarados se exporta correctamente por el hecho de ser **privados**. La solución más fácil: **hacerlos públicos.**

```
type Persona struct {
    Nombre string
    Direccion string
    DNI string
    Edad int
}
```

Pero cómo se convierte a JSON de forma correcta?...

Modelo de datos: serializar a **JSON**

Todos conocemos las ventajas de usar JSON a la hora de enviar y recibir datos, pero en Fabric dicha serialización hay que hacerla a mano. De hecho, nos referiremos como **`marshalling`** y **`unmarshalling`**

```
type Persona
struct {
    Nombre string
    Direccion
string
    DNI string
    Edad int
}
```

```
/*
Converts current Persona to JSON
byte array
*/
func (p *Persona) ToByteArray()
([]byte, error)  {
    if p != nil{
        return json.Marshal(p)
    }
    return nil, fmt.Errorf("p is
null")
}
```

Modelo de datos: escribiendo como **JSON**

Una vez que la información a persistir se tiene serializada como **JSON**, es necesario realizar la operación de escritura siempre y cuando no haya habido errores en los pasos previos.

```
data, err := persona.ToByteArray()
if err == nil {
    err := stub.PutState(persona.DNI, data)
}
```

Ejemplo

```
[ 123456789X, { 'dni': '12345678X', 'nombre': 'Alice', … } ]
```

Fabric SDK

Formato de las respuestas

Formato de las respuestas

Toda transacción que se envíe a la Blockchain de Fabric devuelve como respuesta un objeto del tipo **`peer.Response`**

```
type Response struct {
        // A status code that should follow the HTTP status codes.
        Status int32 `protobuf:"varint,1,opt,name=status" json:"status,omitempty"`
        // A message associated with the response code.
        Message string `protobuf:"bytes,2,opt,name=message" json:"message,omitempty"`
        // A payload that can be used to include metadata with this response.
        Payload []byte `protobuf:"bytes,3,opt,name=payload,proto3" json:"payload,omitempty"`
}
```

A response with a representation similar to an HTTP response that can be used within another message.

Códigos de respuesta

Fabric admite los siguientes códigos de respuesta

- OK = **200** (init or invoke success)
- ERRORTHRESHOLD = **400**
- ERROR = **500** (default error)

peer.Response = SUCCESS

SUCCESS

Para aquellas respuestas que se hayan ejecutado correctamente. La respuesta se enviara con el campo status = 200 (HTTP OK), y el campo payload podrá contener los datos de respuesta que el desarrollador considere necesario.

```
return shim.Success(nil)
```

peer.Response = ERROR

ERROR

Para aquellas respuestas que **NO** se hayan ejecutado correctamente. La respuesta se enviara con el campo **status** = 500 (HTTP ERROR), y el campo **message** tendrá los detalles del error en formato **string**

```
return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
```

Ejemplo respuesta

```
// Invoke is called per transaction on the chaincode. Each transaction is
// either a 'get' or a 'set' on the asset created by Init function. The Set
// method may create a new asset by specifying a new key-value pair.
func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
        // Extract the function and args from the transaction proposal
        fn, args := stub.GetFunctionAndParameters()
        param0 := args[0]
        if param0 == "ethereum mola"{
                return shim.Error(err.Error())
        } else {
                // Return the result as success payload
                msg := "esto es una respuesta valida"
                return shim.Success([]byte(msg))
        }
}
```

Buenas prácticas

Programar `chaincodes` y no morir en el intento

Tips y consejos

Ten en cuenta estos consejos a la hora de programar tu **`smart contract`**

- Testear, testear y testear

- Ejemplo completo de **`TDD`**: https://www.ibm.com/developerworks/cloud/library/cl-ibm-blockchain-chaincode-testing-using-golang/cl-ibm-blockchain-chaincode-testing-using-golang-pdf.pdf
- Y... si algo no existe... se Mockea, el **`stub`**, el **`history`**, etc.

Tips y consejos

Ten en cuenta estos consejos a la hora de programar tu **`smart contract`**

- Los métodos públicos empiezan por mayúscula, los privados por minúscula.
- En GO no existe el concepto de herencia, sólo la composición.
- En una operación de escritura, se pueden modificar todos los datos que se quiera, pero solo se guardará el estado final de los mismos (**`state`**).
- Las operaciones de lectura son muy rápidas porque se hacen contra el propio nodo y no contra el **`ledger`** (y casi instantáneas cuando accedes por clave).
- Recuperar el historial de transacciones no se realiza contra el **`state`**, sino contra el **`ledger`**. Por eso, requiere de un elemento **`iterator`**
- **`Una interfaz`** no se puede usar como puntero

Buenas prácticas

Como sabemos que programar en un lenguaje poco usual como **Go** puede ser algo difícil, a continuación os enseñamos algunas buenas prácticas para **Fabric**.

- Cuidado con los **logs por consola** que pueden generar un **panic** en **GO** y parar por completo la ejecución del smart contract.
- Prohibido usar el nombre **init** como nombre de paquete dentro de un smart contract. Este, está reservado para el SDK de Fabric.
- Si se quiere imprimir por pantalla algo que se desconoce su tipo de dato, serializarlo como **interface{}** e imprimir el resultado
- Realizar una conversión de tipo segura para evitar **panics.**
- Diferenciar la operación **init** y **update** a pesar de que ambas se ejecutan al instanciar un nuevo smart contract. (**init** con la operación **instantiate** y **update** con la operación **update**)
- Si repites el mismo código en varios sitios, usa una **interfaz.**
- Guardar los datos serializados como **JSON o ProtoBuffer**

Fin

Imaginas un **`smart contract`** que sirva para muchos proyectos?